Jaypee University of Information Technology
Solan (H.P.)
LEARNING RESOURCE CENTER

Acc. Num. SP03014 Call Num:

General Guidelines:

- Library books should be used with great care.
- Tearing, folding, cutting of library books or making any marks on them is not permitted and shall lead to disciplinary action.
- Any defect noticed at the time of borrowing books must be brought to the library staff immediately. Otherwise the borrower may be required to replace the book by a new copy.
- The loss of LRC book(s) must be immediately brought to the notice of the Librarian in writing.

Learning Resource Centre-JUIT

SP03014

IPv4-IPv6 INTEROPERABILITY

By

Akhil Bhardwaj 031226

Bhaskar Vijay Singh 031408

Shashank Shah 031201

May-2007

Submitted in partial fulfillment of the Degree of Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY

CERTIFICATE

This is to certify that the work entitled, "IPv4 – IPv6 Interoperability" submitted by Akhil Bhardwaj, Bhaskar Vijay Singh and Shashank Shah in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science and Engineering of Jaypee University of Information Technology has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Brig.(Retd.) S.P. Ghrera

HOD Computer Science and Engineering.

// ACKNOWLEDGEMENT

In Accordance with our final project submission of 8th Semester(B.Tech Computer Science & Engg.), we were assigned to study and research on ongoing transition mechanisms of IPv4 to IPv6 protocol and making an application for same.

We would like to express our extreme gratitude to

Retd. (Brig). S.P. Ghrera

Head Of Department

Computer Science & Engg.

Jaypee University Of Information Technology

For guiding us, being extremely helpful, patient and always being there whenever we were in doubt. Without his help, support and constant supervision, we would have never been able to complete our final project assignment successfully.

CONTENTS

I Certificate 2
II Acknowledgement 3
III List of figures 5
IV List of tables 6
V Synonyms 7
VI Abstract 9
1 Study And Research
  1.1 Introduction 10
    1.1.1 Internet Protocol 10
    1.1.2 Internet Protocol Version 4 12
    1.1.3 Internet Protocol Version 6 16
  1.2 Interoperability
    1.2.1 Introduction 23
    1.2.2 Dual Stack Techniques 25
    1.2.3 Tunneling 26
    1.2.4 Translation 35
  1.3 Selection of transition technique 43
2 Design Of Application
  2.1 Statement of purpose 44
  2.2 Context of the application 44
  2.3 Data flow diagram 44
  2.4 Class and Structure specification 46
  2.5 Flow Charts 51
  2.6 Test Application 58
    2.6.1 Class and Structure specification 58
    2.6.2 Flow Chart 60
3 Implementation
  3.1 Main Application Source Code 62
  3.2 Test Application Source Code 75
4 Experimental setup and testing
  4.1 Setup , test procedure & Results 80
5 Deployment 82
  5.1 Screen Shots 83
    5.1.1 Tunnel Application 83
    5.1.2 Test Application 86
  5.2 Directories and file structure 87
    5.2.1 Tunnel Application 87
    5.2.2 Test Application 88
  5.3 Environment for compilation & linking 88
    5.3.1 Tunnel Application 88
    5.3.2 Test Application 88
6 Conclusion 89
7 Bibliography 90

LIST OF FIGURES

1.	Internet protocols span the complete range of OSI model layers	11
2.	Fourteen fields comprise an IPv4 packet.	12
3.	An IP address consists of 32 bits, grouped into four octets.	15
4.	IP address formats A, B, and C are available for commercial use.	16
5.	IPv6 Header Structure	19
6.	IPv4 vs IPv6 address	21
7.	Different Transition Strategies	24
8.	Dual Stack Model	26
9.	Architecture of Configured Tunnel	28
10.	Typical Use of Configured Tunnel	28
11.	Data stream of Router-to-Router Tunnel	29
12.	Automatic Tunnel Overview	30
13.	Data Stream for Automatic Tunnels	30
14.	6to4 Tunnel Mechanism Architecture	31
15.	The architecture of Tunnel Broker	32
16.	IPv6 address format of ISATAP node	33
17.	DSTM Architecture and working procedure	34
18.	Translation between IPv4 network and IPv6 network	36
19.	Stateless IP/ICMP Translation Algorithm	36
20.	Operation of NAT-PT	37
21.	Application Layer Gateway	38
22.	Operation of TRT	39
23.	Sock64 Architecture	40
24.	Structure of dual stack host with BIS	41
25.	Structure of Dual Stack host with BIA	42
26.	Data Flow Diagram for tunnel application	45
27.	Flow chart of the constructor function of the class CNetWorkInterface	51
28.	Initialize Function of CNetworkInterface	52
29.	Flow Chart of the MainThreadLoop() Function	53
30.	Flow Chart of the function InitializeBtoSThread	54
31.	Flow Chart of the function BtoSthreadLoop	55
32.	Flow Chart of the function HtoBThreadLoop	56
33.	Flow Chart of the function Stop	57
34.	Flow chart of the test application	60
35.	Hardware Setup	80
36.	Deployment of the application in the real world	82
37.	Initial Configuration of the Main Application	83
38.	After the start of the execution of the main application	84
39.	The Status message after the completion of file transfer	85
40.	Both ends of the test application	86

LIST OF TABLES

1 Reference Information About the Five IPv4 Address Classes 15
2 Classes of IPv4 to IPv6 migration mechanisms 24

SYNONYM

IPv4-compatible address: IPv4-compatible address is identified by an all-zero 96-bit prefix and an IPv4 address in the low-order 32 bits. It can be written as ::[IPv4] IPv4-mapped address: IPv4- mapped address is identified as ::FFFF:a.b.c.d, a.b.c.d is IPv4 address.

Dual Stack: A mechanism that supports both IPv4 and IPv6 on hosts or routers.

Tunneling: A mechanism in which one type of packets (e.g. IPv6) are encapsulated inside another type of packets (e.g. IPv4).

Translation: A mechanism that establishes the communication between IPv4 nodes and IPv6 nodes.

Configured tunnel: A kind of point-to-point tunnel, which is configured manually.

Automatic tunnel: A kind of tunnel which is configured and established automatically when necessary and broken up automatically when no longer necessary.

6to4: A semi-automatic tunneling mechanism between routers, which uses 2002:V4ADDR::/48 as prefix.

Tunnel broker: A semi-automatic tunneling mechanism which can help user collect necessary information and interactively set up IPv6 over IPv4 tunnels.

TSP: Tunnel setup protocol.

Teredo: A tunneling mechanism that enables nodes behind IPv4 NATs to obtain IPv6 connectivity by tunneling packets over UDP.

BGP-tunnel: A tunneling mechanism that interconnects IPv6 islands over IPv4 clouds using BGP.

6over4: A tunneling mechanism that allows isolated IPv6 hosts, located on a physical link which has no directly connected IPv6 router, to become fully functional IPv6 hosts by using an IPv4 multicast domain as their virtual local link.

ISATAP: Intra-Site Automatic Tunnel Addressing Protocol, which automatically connects isolated IPv6 hosts within a site via automatic IPv6- in-IPv4 tunnel.

DSTM: Dual Stack Transition Mechanism, which uses IPv4-over-IPv6 tunnel to carry IPv4 traffic within an IPv6-only network and provides a method to allocate a temporary IPv4 Address to a IPv6/IPv4 node.

SIIT: Stateless IP/ICMP Translation Algorithm, which translates between IPv4 header and IPv6 header.

NAT-PT: Network Address Translation with Protocol Translation, which translates an IPv4 packet into a semantically equivalent IPv6 packet and vice versa.

TRT: Transport Relay Translator, a translator which locates in transport layer.

Socks: A system which accepts enhanced IPv4 socks connections from IPv4 hosts and relays them to IPv4 or IPv6 nodes.

ALG: Application Layer Gateway, which allows users behind gateways or firewalls to use applications that otherwise are not allowed to traverse gateways and firewalls

BIS: Bump in the Stack, which is a translation interface between IPv4 applications and IPv6 network infrastructure.

BIA: Bump- in-the-API, which is a translation interface between socket API and TCP/IP modules.

ABSTRACT

IPv6 is a new version of the internetworking protocol designed to address the scalability and service shortcomings of the current standard, IPv4.

Unfortunately, IPv4 and IPv6 are not directly compatible, so programs and systems designed to one standard can not communicate with those designed to the other. IPv4 systems, however, are ubiquitous and are not about to go away "over night" as the IPv6

Systems are rolled in. Consequently, it is necessary to develop smooth transition mechanisms that enable applications to continue working while the network is being upgraded. In this paper we have presented the various transition mechanisms and implementation of a transparent transition service.

As a result, we are able to demonstrate and measure a working system, and report on the complexities involved in building and deploying such a system.

1 STUDY AND RESEARCH

1.1 INTRODUCTION

1.1.1 Internet protocol

Background

The Internet protocols are the world's most popular open-system (nonproprietary) protocol suite because they can be used to communicate across any set of interconnected networks and are equally well suited for LAN and WAN communications. The Internet protocols consist of a suite of communication protocols, of which the two best known are the Transmission Control Protocol (TCP) and the Internet Protocol (IP). The Internet protocol suite not only includes lower-layer protocols (such as TCP and IP), but it also specifies common applications such as electronic mail, terminal emulation, and file transfer. This chapter provides a broad introduction to specifications that comprise the Internet protocols. Discussions include IP addressing and key upper-layer protocols used in the Internet. Specific routing protocols are addressed individually later in this document.

Internet protocols were first developed in the mid-1970s, when the Defense Advanced Research Projects Agency (DARPA) became interested in establishing a packet-switched network that would facilitate communication between dissimilar computer systems at research institutions. With the goal of heterogeneous connectivity in mind, DARPA funded research by Stanford University and Bolt, Beranek, and Newman (BBN). The result of this development effort was the Internet protocol suite, completed in the late 1970s.

TCP/IP later was included with Berkeley Software Distribution (BSD) UNIX and has since become the foundation on which the Internet and the World Wide Web (WWW) are based.

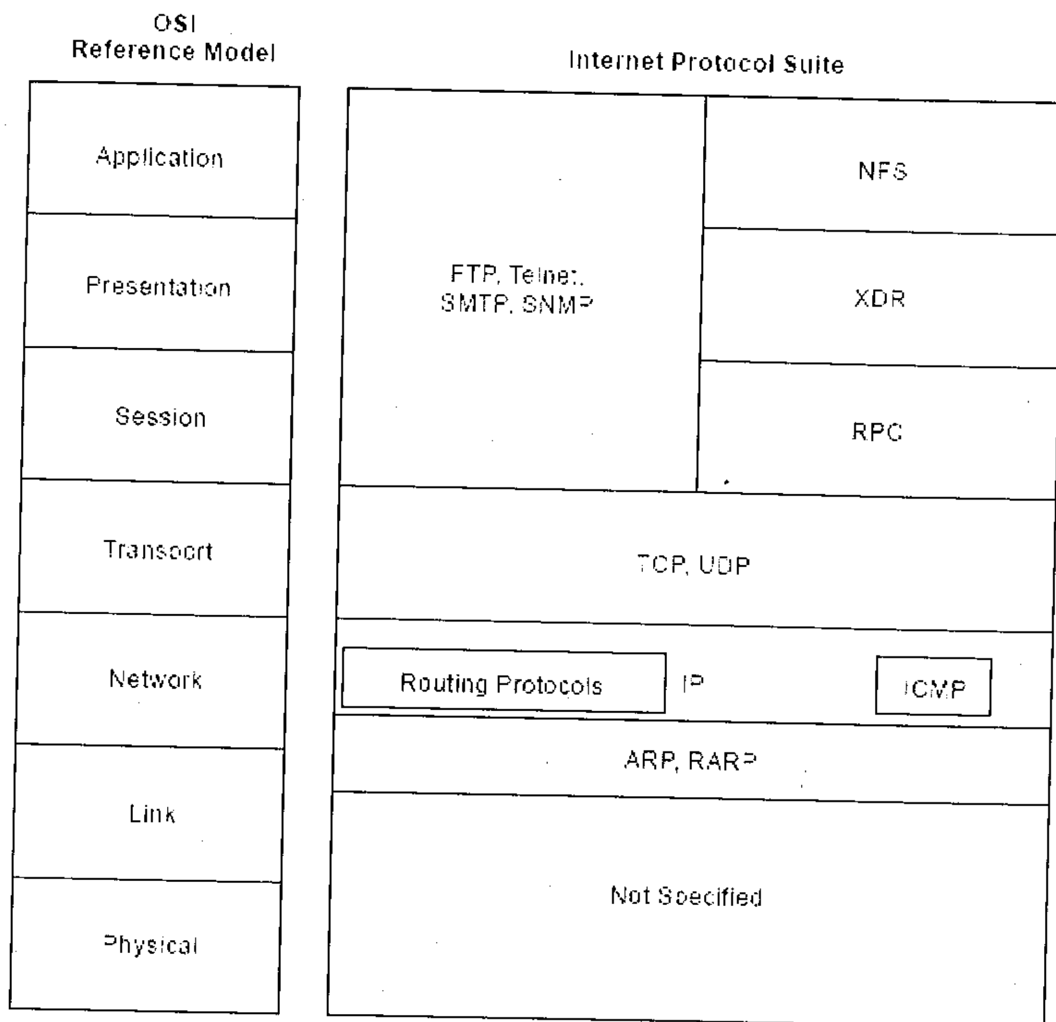


Figure 1 : Internet protocols span the complete range of OSI model layers.

IP is a data oriented protocol used for communicating data across a packet switched network. It provides a service of communicable, unique, global addressing among computers. Data to be transferred is encapsulated in datagrams / packets. No communication link needs to be established between the two communicating devices therefore it is a connectionless protocol.

IP provides a unreliable service since there are no guarantees about the packet it may lead to :

- Data corruption
- Out of order packets
- Duplicate arrival
- Drop Packets

1.1.2 Internet Protocol Version 4

The Internet Protocol (IP) is a network-layer (Layer 3) protocol that contains addressing information and some control information that enables packets to be routed. IP is documented in RFC 791 and is the primary network-layer protocol in the Internet protocol suite. Along with the Transmission Control Protocol (TCP), IP represents the heart of the Internet protocols. IP has two primary responsibilities: providing connectionless, best-effort delivery of datagrams through an internetwork; and providing fragmentation and reassembly of datagrams to support data links with different maximum-transmission unit (MTU) sizes.

IPv4 Packet Format

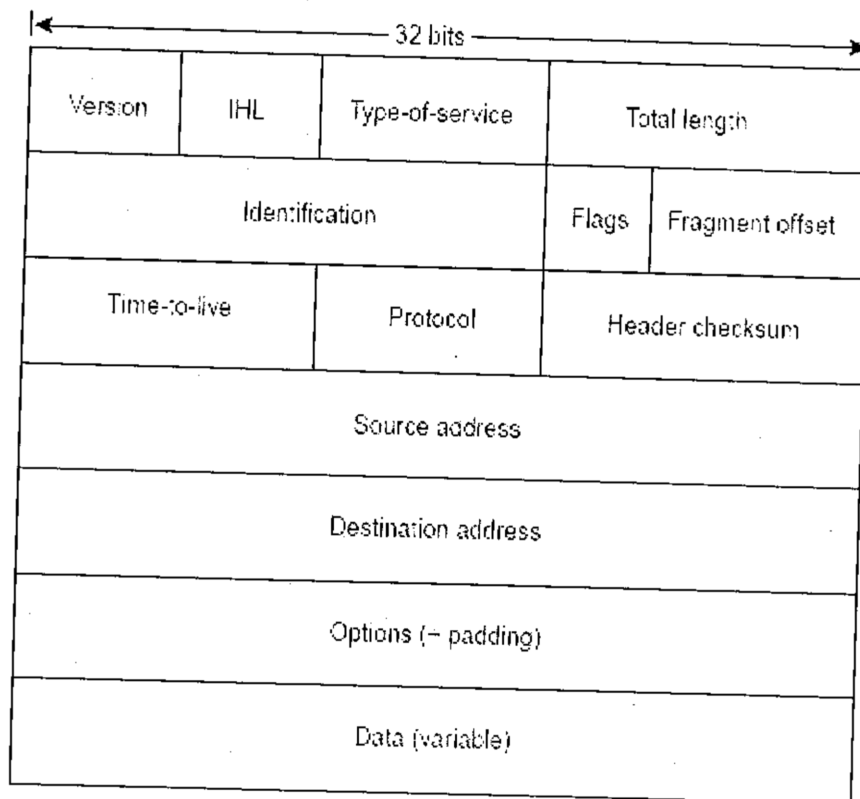


Figure 2 : Fourteen fields comprise an IPv4 packet.

1. *Version*—Indicates the version of IP currently used.

2. *IP Header Length (*IHL)—Indicates the datagram header length in 32-bit words.

3. *Type-of-Service*—Specifies how an upper-layer protocol would like a current datagram to be handled, and assigns datagrams various levels of importance.

4. *Total Length*—Specifies the length, in bytes, of the entire IP packet, including the data and header.

5. *Identification*—Contains an integer that identifies the current datagram. This field is used to help piece together datagram fragments.

6. *Flags*—Consists of a 3-bit field of which the two low-order (least-significant) bits control fragmentation. The low-order bit specifies whether the packet can be fragmented. The middle bit specifies whether the packet is the last fragment in a series of fragmented packets. The third or high-order bit is not used.

7. *Fragment Offset*—Indicates the position of the fragment's data relative to the beginning of the data in the original datagram, which allows the destination IP process to properly reconstruct the original datagram.

8. *Time-to-Live*—Maintains a counter that gradually decrements down to zero, at which point the datagram is discarded. This keeps packets from looping endlessly.

9. *Protocol*—Indicates which upper-layer protocol receives incoming packets after IP processing is complete.

10. *Header Checksum*—Helps ensure IP header integrity.

11. S*ource Address*—Specifies the sending node.

12. *Destination Address*—Specifies the receiving node.

13. *Options*—Allows IP to support various options, such as security.

14. *Data*—Contains upper-layer information.

IPv4 Addressing

As with any other network-layer protocol, the IP addressing scheme is integral to the process of routing IP datagrams through an internetwork. Each IP address has specific components and follows a basic format. These IP addresses can be subdivided and used to create addresses for subnetworks.

Each host on a TCP/IP network is assigned a unique 32-bit logical address that is divided into two main parts: *the network number* and the *host number.* The network number identifies a network and must be assigned by the Internet Network Information Center (InterNIC) if the network is to be part of the Internet. An Internet Service Provider (ISP) can obtain blocks of network addresses from the InterNIC and can itself assign address space as necessary. The host number identifies a host on a network and is assigned by the local network administrator.

IPv4 Address Format

The 32-bit IP address is grouped eight bits at a time, separated by dots, and represented in decimal format (known as *dotted decimal notation*). Each bit in the octet has a binary weight (128, 64, 32,16, 8, 4, 2, 1). The minimum value for an octet is 0, and the maximum value for an octet is 255.

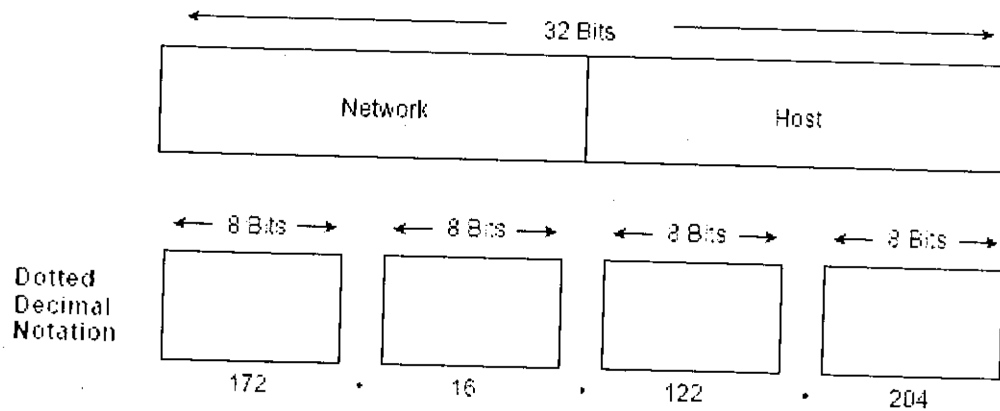


Figure 3 : An IP address consists of 32 bits, grouped into four octets.

IPv4 Address Classes

IP addressing supports five different address classes: A, B,C, D, and E. Only classes A, B, and C are available for commercial use. The left-most (high-order) bits indicate the network class. The following table provides reference information about the five IP address classes.

IP Address Class	Format	Purpose	High-Order Bit(s)	Address Range	No. Bits Network/Host	Max. Hosts
A	N.H.H.H	Few large organizations	0	1.0.0.0 to 126.0.0.0	7/24	16777214 (2^{24}-2)
B	N.N.H.H	Medium size organizations	1, 0	128.1.0.0 to 191.254.0.0	14/16	65534 (2^{16}-2)
C	N.N.N.H	Relatively small organizations	1, 1, 0	192.0.1.0 to 223.255.254.0	21/8	254 (2^{8}-2)
D	N/A	Multicast groups	1, 1, 1, 0	224.0.0.0 to 239.255.255.255	N/A (not for commercial use)	N/A
E	N/A	Experimental	1, 1, 1, 1	240.0.0.0 to 254.255.255.255.255	N/A	N/A

Table 1 : Reference Information About the Five IPv4 Address Classes

The class of address can be determined easily by examining the first octet of the address and mapping that value to a class range in the following table. In an IP address of 172.31.1.2, for example, the first octet is 172. Because 172 falls between 128 and 191, 172.31.1.2 is a Class B address. The following figure summarizes the range of possible values for the first octet of each address class.

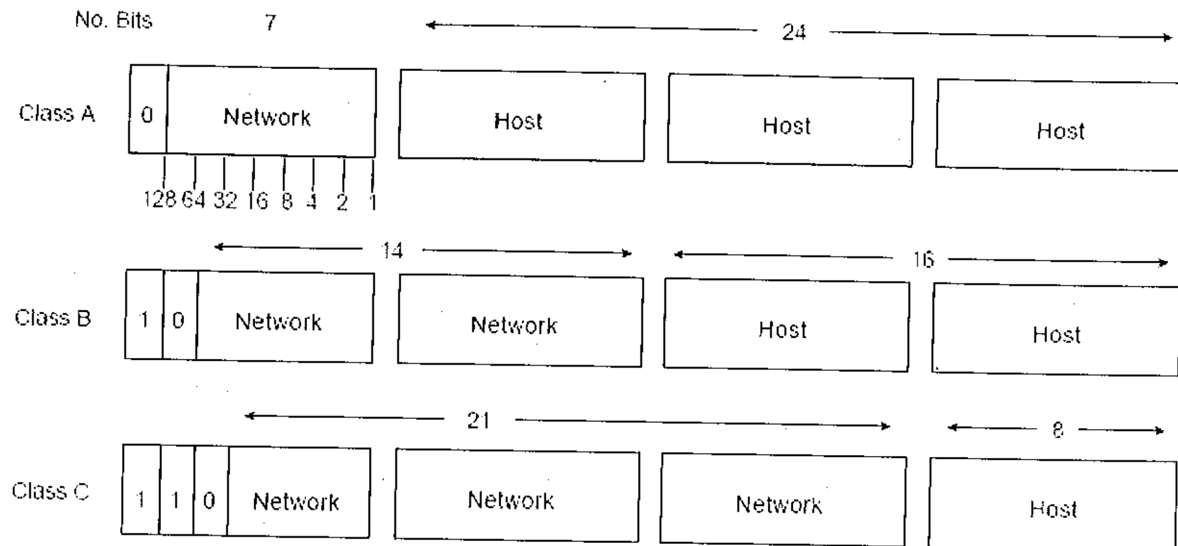


Figure 4 : IP address formats A, B, and C are available for commercial use.

1.1.3 Internet Protocol Version 6

This is the second version to be formally adopted and the main changes from IPv4 are:

- Expanded Addressing capability: The address size of IPv6 has increased from 32bits to 128 bits.
- Simplification of header
- Improved support for extensions and headers.
- Extensions for authentication and privacy
- Flow labeling Capability

The fields that have been removed from IPv4 header:

- *Header Length*: Not needed as the header length is fixed.

- *Identification field*, *Flags* (DF& MF) and *offset field* were removed because Fragmentation is handled only in the sending host in IPv6: routers never fragment a packet, and hosts are expected to use PMTU discovery.
- *Header Checksum*: To remove the processing speed as every router does not calculate the checksum. The checksum is calculated at the upper layers of the protocol.

The fields of IPv6 header different from IPv4 are:

- *Flow Label*: 20 bits field. It distinguishes packets which require similar treatment and routers process these packets more efficiently because they need not process the IP header again and again. All packets belonging to the same flow must have the same source and destination address.
- *Payload length*: 16 bits field specifying the length of the data carried .It limits the maximum packet pay load size to 64KB and IPv6 has a jumbogram extension header which supports bigger packet sizes.
- *Next Header*: 8 bits and is same as Protocol field of the IPv4 header.
- *Hop Limit*: 8 bits and is same as the TTL field of IPv4.
- *Source Address*: 128 bits.
- *Destination Address*: 128 bits.

Listed below is an overview of several features and benefits IPv6 is intended to provide.

- *Larger address space* – IPv6 increases the IP address size from 32 bits to 128 bits. Increasing the size of the address field increases number of unique IP addresses from approximately 4,300,000,000 (4.3×10^9) to 340,282,366,920,938,463,463,374,607,431,768,211,456 (3.4×10^{38}).

  The bigger address space IPv6 offers is the most obvious enhancement it has over IPv4. While today's Internet architecture is based on 32-bit wide addresses, the new version has 128-bit technology available for addressing. Thanks to the enlarged address space, workarounds like NAT don't have to be used anymore. This allows full, unconstrained IP connectivity for today's IP-based machines as well as upcoming mobile devices like PDAs and cell phones -- all will benefit from full IP access through GPRS and UMTS.

Increasing the address space to 128 bits provides the following additional potential benefits:

- Enhanced applications functionality – Simplifies direct peer-to-peer applications and networking by providing a unique address to each device.
- End-to-end transparency – The increased number of available addresses reduce the need to use address translation technologies
- Hierarchical addressing – The hierarchical addressing scheme provides for address summarization and aggregation. These approaches simplify routing and manage routing table growth.
- Auto-configuration – Clients using IPv4 addresses use the Dynamic Host Configuration Protocol (DHCP) server to establish an address each time they log into a network. This address assignment process is called stateful auto-configuration. IPv6 supports a revised DHCPv6 protocol that supports stateful auto-configuration, and supports stateless auto-configuration of nodes. Stateless auto-configuration does not require a DHCP server to obtain addresses. Stateless auto-configuration uses router advertisements to create a unique address. This creates a "plug-and-play" environment, simplifying address management and administration. IPv6 also allows automatic address configuration and reconfiguration. This capability allows administrators to re-number network addresses without accessing all clients.
- Scalability of multicast routing – IPv6 provides a much larger pool of multicast addresses with multiple scoping options.

- *Mobility* - When mentioning mobile devices and IP, it's important to note that a special protocol is needed to support mobility, and implementing this protocol -- called "Mobile IP" -- is one of the requirements for every IPv6 stack. Thus, if you have IPv6 going, you have support for roaming between different networks, with global notification when you leave one network and enter the other one. Support for roaming is possible with IPv4 too, but there are a number of hoops that need to be jumped in order to get things working. With IPv6, there's no need for this, as support for mobility was one of the design requirements for IPv6.

- *Security* - Besides support for mobility, security was another requirement for the successor to today's Internet Protocol version. As a result, IPv6 protocol stacks are required to include IPsec. IPsec allows authentication, encryption, and compression of IP traffic. Except for application-level protocols like SSL or SSH, all IP traffic between two nodes can be handled without adjusting any applications. The benefit of this is that all applications on a machine can benefit from encryption and authentication, and that policies can be set on a per-host (or even per-network) basis, not per application/service.

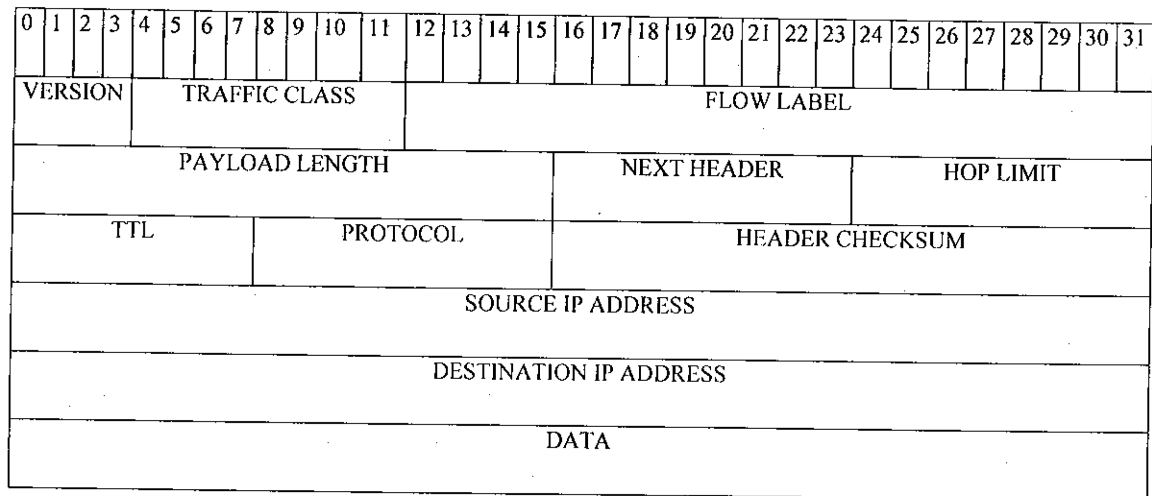


Figure 5 : IPv6 Header Structure

1. *Version.* 4 bits. IPv6 version number.
2. *Traffic Class.* 8 bits. Internet traffic priority delivery value.
3. *Flow Label.* 20 bits. Used for specifying special router handling from source to destination(s) for a sequence of packets.
4. *Payload Length.* 16 bits unsigned. Specifies the length of the data in the packet. When cleared to zero, the option is a hop-by-hop Jumbo payload.

5. *Next Header.* 8 bits. Specifies the next encapsulated protocol. The values are compatible with those specified for the IPv4 protocol field.
6. *Hop Limit.* 8 bits unsigned. For each router that forwards the packet, the hop limit is decremented by 1. When the hop limit field reaches zero, the packet is discarded. This replaces the TTL field in the IPv4 header that was originally intended to be used as a time based hop limit.
7. *Source address.* 16 bytes. The IPv6 address of the sending node.
8. *Destination address.* 16 bytes. The IPv6 address of the destination node.

IPv6 Addressing

IPv6 addresses use 128-bit technology, which results in 2^{128} theoretically addressable hosts. This allows a *really* big number of machines to be addressed, and it will fit all today's requirements plus PDAs, cell phones, and even IP phones in the near future without any problem. When writing IPv6 addresses, they are usually divided into groups of 16 bits written as four hex digits, and the groups are separated by colons. An example is:

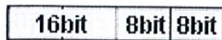
fe80::2a0:d2ff:fea5:e9f5

This shows a special thing -- a number of consecutive zeros can be abbreviated by a single "::" once in the v6 number. The above address is thus equivalent to fe80:0:00:000:2a0:d2ff:fea5:e9f5 -- leading zeros within groups can be omitted.

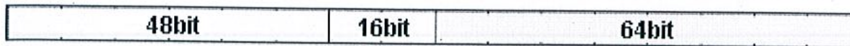
To make addresses manageable, they are split in two parts, which are the bits identifying the network a machine is on, and the bits that identify a machine on a network or subnetwork. The bits are known as netbits and hostbits, and in both IPv4 and v6, the netbits are the "left," or most significant bits of an IP number; and the host bits are the "right," or least significant bits:

N netbits | 128-n hostbits

IPv4:



IPv6:



Provider-assigned network-bits

Self-assigned subnet-bits

Host-bits

Figure 6 : IPv4 vs IPv6 address

IPv6 addresses have a similar structure to class B addresses.

Now while the space for network and subnets is sufficient, using 64 bits for addressing hosts seems like a waste. It's unlikely that you will want to have several billion hosts on a single subnet, so what is the idea behind this?

The idea behind having fixed-width, 64-bit wide host identifiers is that they aren't assigned manually as in IPv4. Instead, v6 host addresses are recommended to be built from so-called EUI64 addresses. EUI64 addresses are -- as the name says -- 64-bits wide, and derived from MAC addresses of the underlying network interface. For example, with Ethernet, the 6-byte (48-bit) MAC address is usually filled with the hex bits "fffe" in the middle -- the MAC address

01:23:45:67:89:ab

results in the EUI64 address

01:23:45:ff:fe:67:89:ab

which again gives the host bits for the IPv6 address.

::0123:45ff:fe67:89ab

These host bits can now be used to automatically assign IPv6 addresses to hosts, which supports autoconfiguration of v6 hosts -- all that's needed to get a complete v6 IP number is the first (net/subnet) bits. IPv6 also offers a solution to assign them automatically.

When on a network of machines speaking IP, there's usually one router which acts as the gateway to outside networks. In IPv6 land, this router will send "router advertisement" information which clients are expected to either receive during operation or solicit upon startup. The router advertisement information includes data on the router's address, and which address prefix it routes. With this information and the host-generated EUI64 address, a v6-host can calculate its IP number, and there is no need for manual address assignment. Of course, routers still need some configuration.

The advertisement information routers create is part of the Neighbor Discovery Protocol (NDP, see [RFC2461]), which is the successor to IPv4's ARP protocol. In contrast to ARP, NDP does not only do lookup of v6 addresses for MAC addresses (the neighbor solicitation/advertisement part), but also does a similar service for routers and the prefixes they serve, which is used for autoconfiguration of v6 hosts.

1.2 INTEROPERABILITY

1.21 Introduction

IPv6 is the proposed replacement to IPv4 and modifies the unsuitable and redundant parts of the original and replaces them with more suitable features for its assumed role. Primarily this meant increasing the address space from 32 to 128 bits, more than enough for any perceived future usage. Also IPv6 improves the header system, removing the redundant parts and defining extension headers to make processing more efficient. Unfortunately, due (primarily) to addressing issues, IPv4 and IPv6 are not compatible. Due to this, and the enormous task of converting all the IPv4 systems in the world to IPv6, deployment of IPv6 will prove to be a rather complicated affair.

The process of transition between IPv4 and IPv6 is a major issue in the networking community. While the deployment of IPv6 would be facilitated by the deployment of IPv6 services and applications, suppliers are reluctant to fully support IPv6 waiting first to see if there is sufficient interest. Unfortunately, users are reluctant to deploy IPv6 until they can get a comparable level of service to that in their IPv4 networks. In an attempt to resolve the situation and increase the speed of IPv6 deployment, a number of transition tools have been developed to ease the process of transition and allow users to deploy IPv6 in a useful way. These can vary from simple techniques to quite complicated mechanisms.

IPv6 and IPv4 will coexist for many years. A wide range of techniques has therefore been defined that make the coexistence possible and provide an easy transition. There are three main categories:

- Dual-stack techniques allow IPv4 and IPv6 to coexist in the same devices and networks.
- Tunneling techniques allow the transport of IPv6 traffic over the existing IPv4 infrastructure.
- Translation techniques allow IPv6-only nodes to communicate with IPv4-only nodes.

These techniques can and likely will be used in combination with one another. The migration to IPv6 can be done step by step, starting with a single host or subnet.

The transition mechanisms can be broadly classified into three categories:

Dual stack	Tunnelling	Translation
	• Configured Tunnels	• SIIT
	• Automatic Tunnels	• NAT-PT
	• 6to4	• TRT
	• Tunnel Broker	• Socks
	• TSP	• ALG
	• Teredo	• BIS
	• BGP Tunnel	• BIA
	• 6over4	
	• ISATAP	
	• DSTM	

Table 2 : Classes of IPv4 to IPv6 migration mechanisms

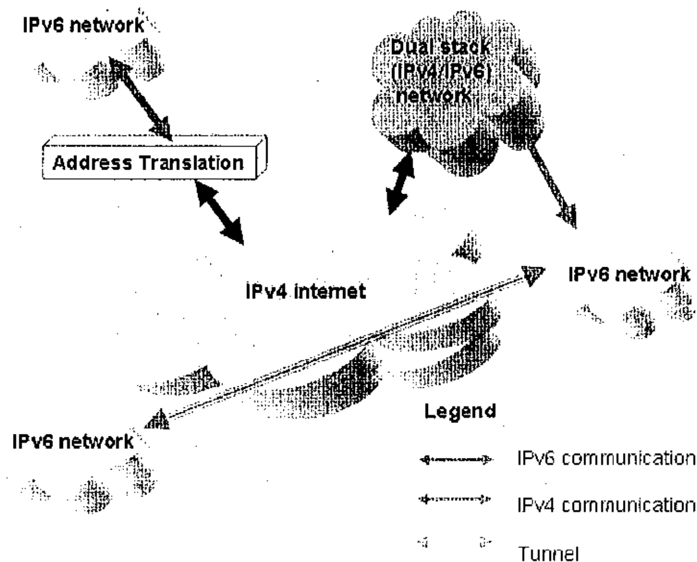


Figure 7 : Different Transition Strategies

1.2.2 Dual Stack Techniques

A dual-stack node has complete support for both protocol versions. This type of node is often referred to as an IPv6/IPv4 node. In communication with an IPv6 node, such a node behaves like an IPv6-only node, and in communication with an IPv4 node, it behaves like an IPv4-only node. Implementations probably have a configuration switch to enable or disable one of the stacks. So this node type can have three modes of operation.

- When the IPv4 stack is enabled and the IPv6 stack is disabled, the node behaves like an IPv4-only node.
- When the IPv6 stack is enabled and the IPv4 stack disabled, it behaves like an IPv6-only node.
- When both the IPv4 and IPv6 stacks are enabled, the node can use both protocols. An IPv6/IPv4 node has at least one address for each protocol version. It uses IPv4 mechanisms to be configured for an IPv4 address (static configuration or DHCP) and uses IPv6 mechanisms to be configured for an IPv6 address (static configuration or autoconfiguration).

DNS is used with both protocol versions to resolve names and IP addresses. An IPv6/IPv4 node needs a DNS resolver that is capable of resolving both types of DNS address records. The DNS A record is used to resolve IPv4 addresses and the DNS AAAA or A6 record is used to resolve IPv6 addresses.

In some cases, DNS returns only an IPv4 or an IPv6 address. If the host that is to be resolved is a dualstack host, DNS might return both types of addresses. Hopefully, for this case, both the DNS resolver on the client and an application using DNS will have configuration options that let us specify orders or filters of how to use the addresses (i.e., preferred protocol settings). Generally, applications that are written to run on dual-stack nodes need a mechanism to determine whether it is communicating with an IPv6 peer or an IPv4 peer. Note that the DNS resolver may run over an IPv4 or IPv6 network, but the worldwide DNS tree is mainly reachable through an IPv4 network layer.

A dual-stack network is an infrastructure in which both IPv4 and IPv6 forwarding is enabled on routers.

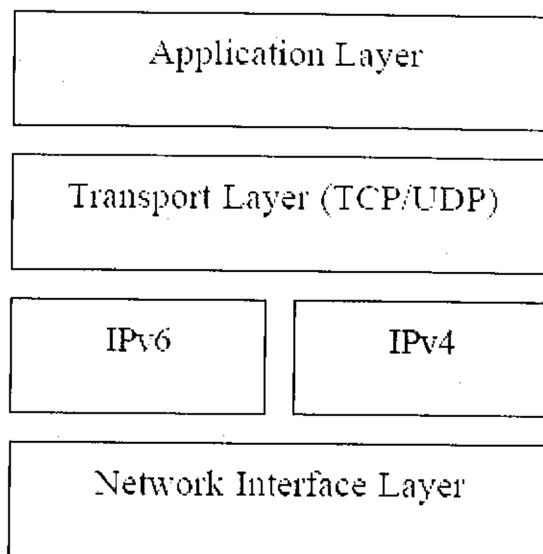


Figure 8 : Dual Stack Model

The disadvantage of this technique is that you must perform a full network software upgrade to run the two separate protocol stacks. This means all tables (e.g., routing tables) are kept simultaneously, routing protocols being configured for both protocols. For network management, you have separate commands, depending on the protocol (e.g., ping.exe for IPv4 and ping6.exe for IPv6 on a host with a Microsoft operating system—both commands with different command options), and it takes more memory and CPU power.

1.2.3 Tunnelling

Tunneling is a mechanism that one type of packets is encapsulated inside another type of packets. In the case of IPv4 to IPv6 migration, we can encapsulate IPv6 packets inside IPv4 packets. This enables the current IPv4 infrastructure to carry IPv6 packets. This mechanism is especially important for IPv4 to IPv6 migration, since the existing internet is based upon an IPv4 infrastructure. Many of the transition mechanisms we will talk about later are based on tunneling. There are four possible tunnel types that could be established between routers and hosts:

- Router to router: An IPv6/IPv4 router tunnels IPv6 packets to another IPv6/IPv4 router via IPv4 infrastructure.
- Host to router: An IPv6/IPv4 host tunnels IPv6 packets to an IPv6/IPv4 router via IPv4 infrastructure.
- Host to host: An IPv6/IPv4 host tunnels IPv6 packets to another IPv6/IPv4 host viaIPv4 infrastructure.
- Router to host: An IPv6/IPv4 router tunnel IPv6 packets to an IPv6/IPv4 host, which is the final destination.

Configured Tunnels

Configured tunnels are point-to-point and manually configured tunnels, which be used to connect IPv6 hosts or networks over an IPv4 infrastructure. The IPv4 address of tunnel endpoint is determined by configuration information on the encapsulating node. Therefore, the encapsulating node must keep the information about the addresses of all the tunnel endpoints.

Configured tunnels are normally used between sites where traffic will be exchanged regularly. Configured tunnels are advantageous over automatic tunnels for control of the tunnel paths, and to reduce the possibility of tunnel relay denial-of-service attacks.

We can set up a direct tunnel with each site that we need to access. However, this way we have to manage a lot of tunnels. An efficient way is that we set up a direct tunnel with an ISP which can guarantee IPv6 connectivity to IPv6 world. For example, we set up a tunnel with 6Bone and thus obtain IPv6 connectivity to other IPv6 sites.

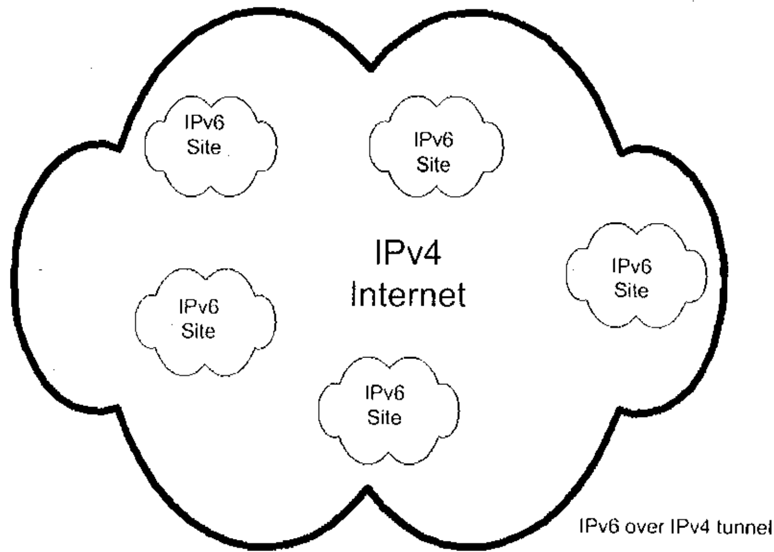


Figure 9 : Architecture of Configured Tunnel

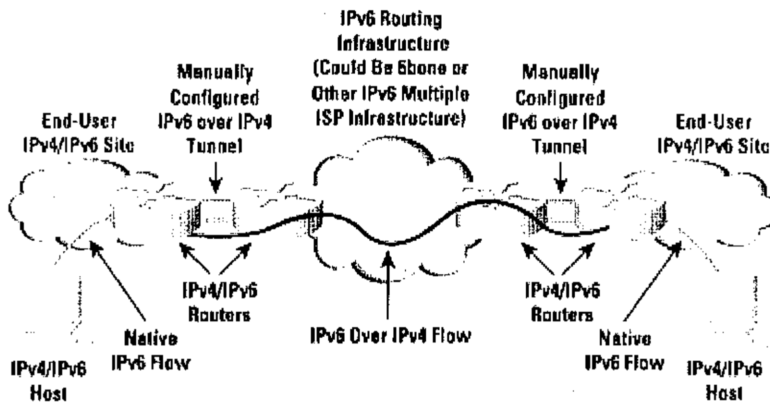


Figure 10 : Typical Use of Configured Tunnel

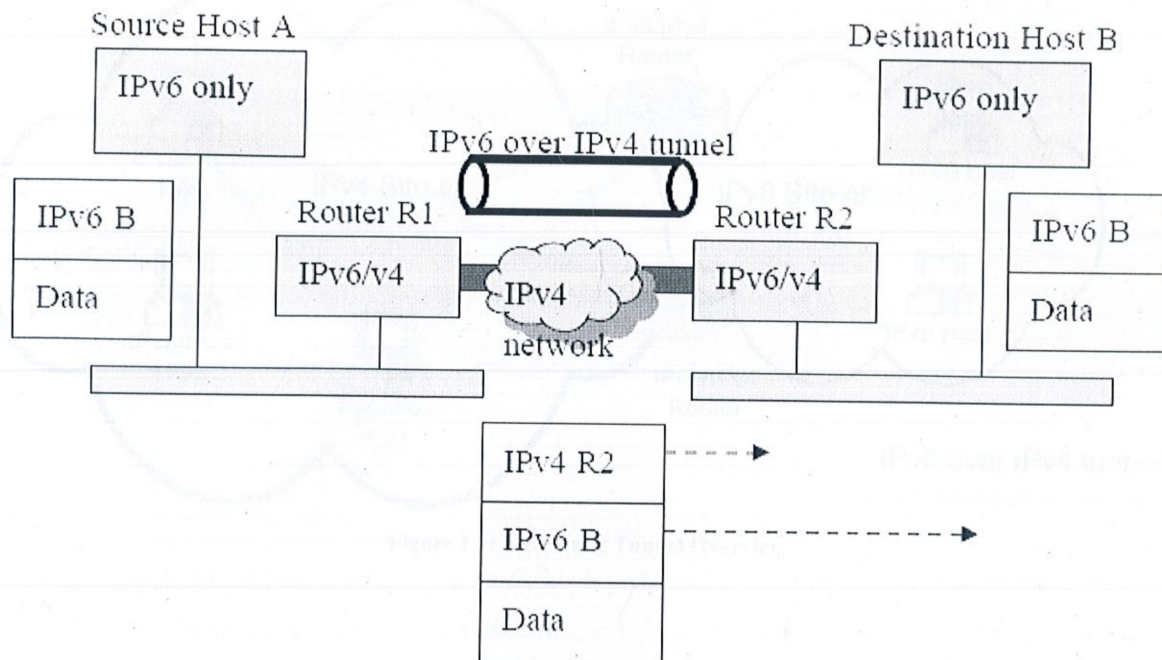


Figure 11 : Data stream of Router-to-Router Tunnel

Automatic Tunnel

Automatic tunnels are tunnels which are automatically created when needed and broken up when no longer needed. The IPv6 address of tunnel end point is derived from its IPv4 which is called IPv4 compatible address, written as ::[IPv4 address]. This allows the host to get the IPv4 address of tunnel end point from its IPv6 address. Automatic tunnels are normally used between hosts or between networks where there is incidental traffic exchange. Because automatic tunnels use IPv4-compatible addresses, they are bad for the IPv6 address architecture. Therefore, automatic tunnels are deprecated.

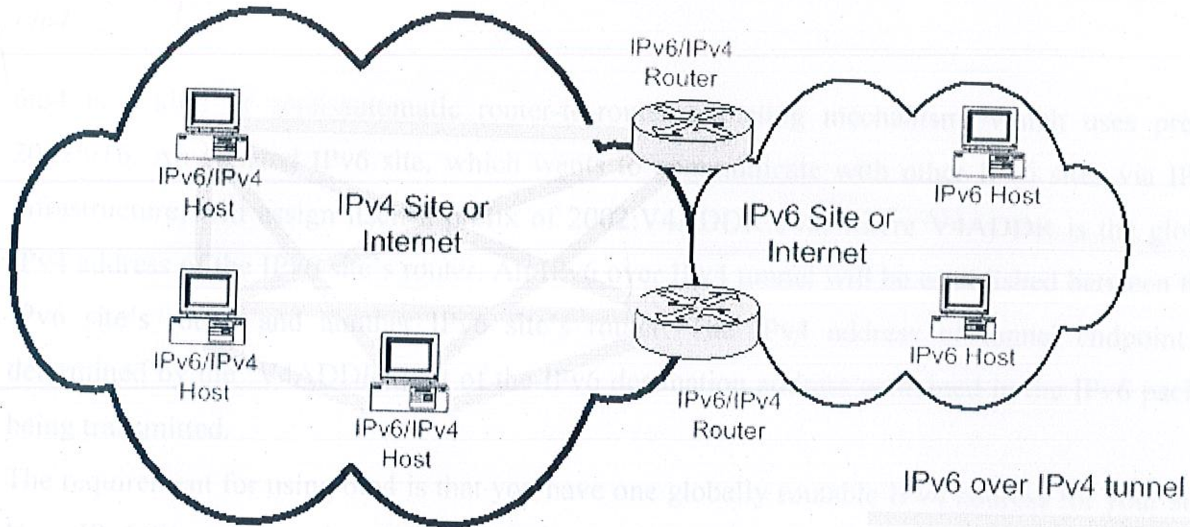


Figure 12 : Automatic Tunnel Overview

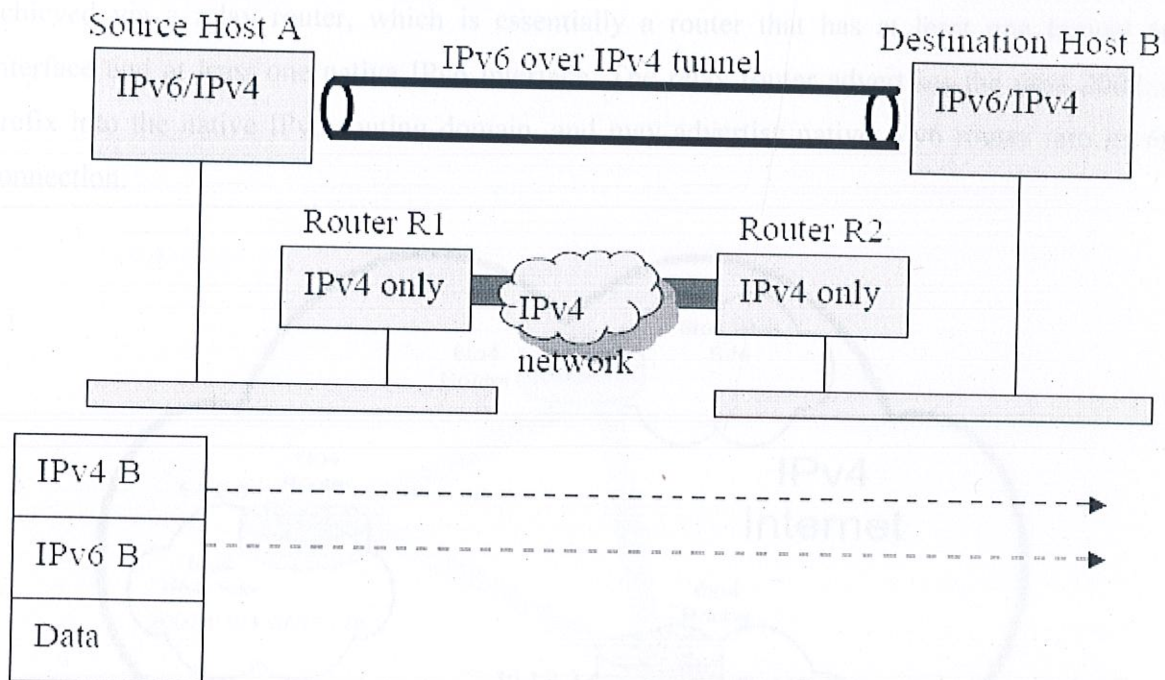


Figure 13 : Data Stream for Automatic Tunnels

6to4

6to4 is a kind of semi-automatic router-to-router tunneling mechanism, which uses prefix 2002::/16. An isolated IPv6 site, which wants to communicate with other IPv6 sites via IPv4 infrastructure, will assign itself a prefix of 2002:V4ADDR::/48, where V4ADDR is the global IPv4 address of the IPv6 site's router. An IPv6 over IPv4 tunnel will be established between this IPv6 site's router and another IPv6 site's router. The IPv4 address of tunnel endpoint is determined by the 'V4ADDR' part of the IPv6 destination address contained in the IPv6 packet being transmitted.

The requirement for using 6to4 is that you have one globally routable IPv4 address for your site. Your IPv6 site may consist of several IPv6 machines. The globally routable IPv4 address must be the IPv4 address of the 6to4 router in your IPv6 site. There is a special situation, when 6to4 site wants to communicate with IPv6-only site. In this case, communication between the sites is achieved via a relay router, which is essentially a router that has at least one logical 6to4 interface and at least one native IPv6 interface. The relay router advertises the 6to4 2002::/16 prefix into the native IPv6 routing domain, and may advertise native IPv6 routes into its 6to4 connection.

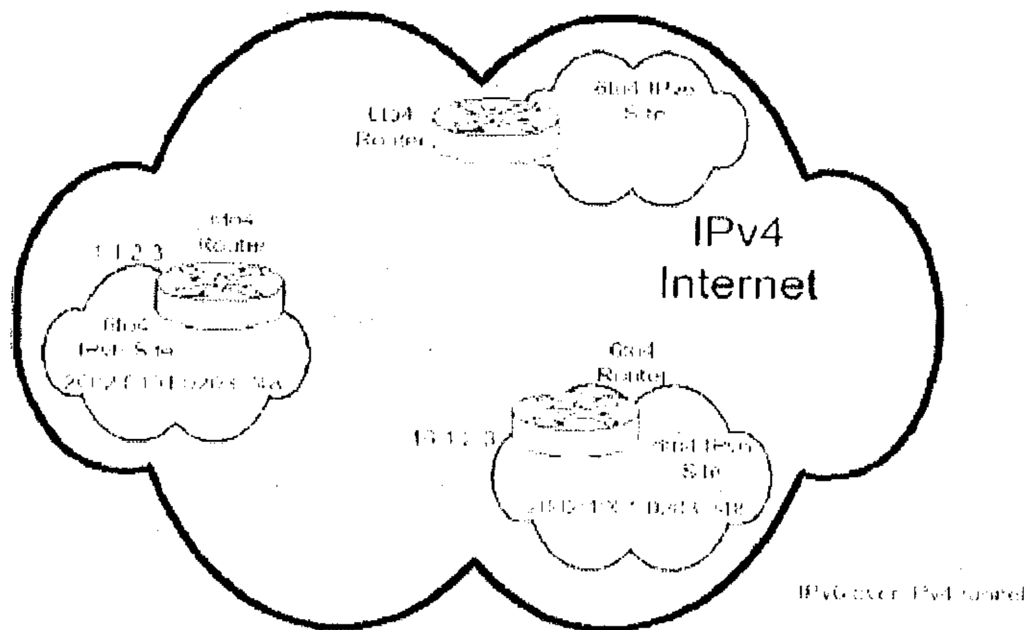


Figure 14 : 6to4 Tunnel Mechanism Architecture

Tunnel Broker

Tunnel broker is a semi-automatic tunneling mechanism. Configuring tunnels requires cooperation of the two parties to set up the correct tunnels. Tunnel broker can be used to help people to implement this. Tunnel broker can be looked as an IPv6 ISP offering IPv6 connectivity through IPv6 over IPv4 tunnels. Current implementations are web-based tools, which allow interactive setup of an IPv6 over IPv4 tunnel. By requesting a tunnel, the tunnel client gets assigned IPv6 addresses out of the address space of the tunnel provider, which can be a single address or a network prefix. The created tunnel is between the tunnel client and the tunnel server. Through the tunnel server, the tunnel client can get connected to IPv6 internet.

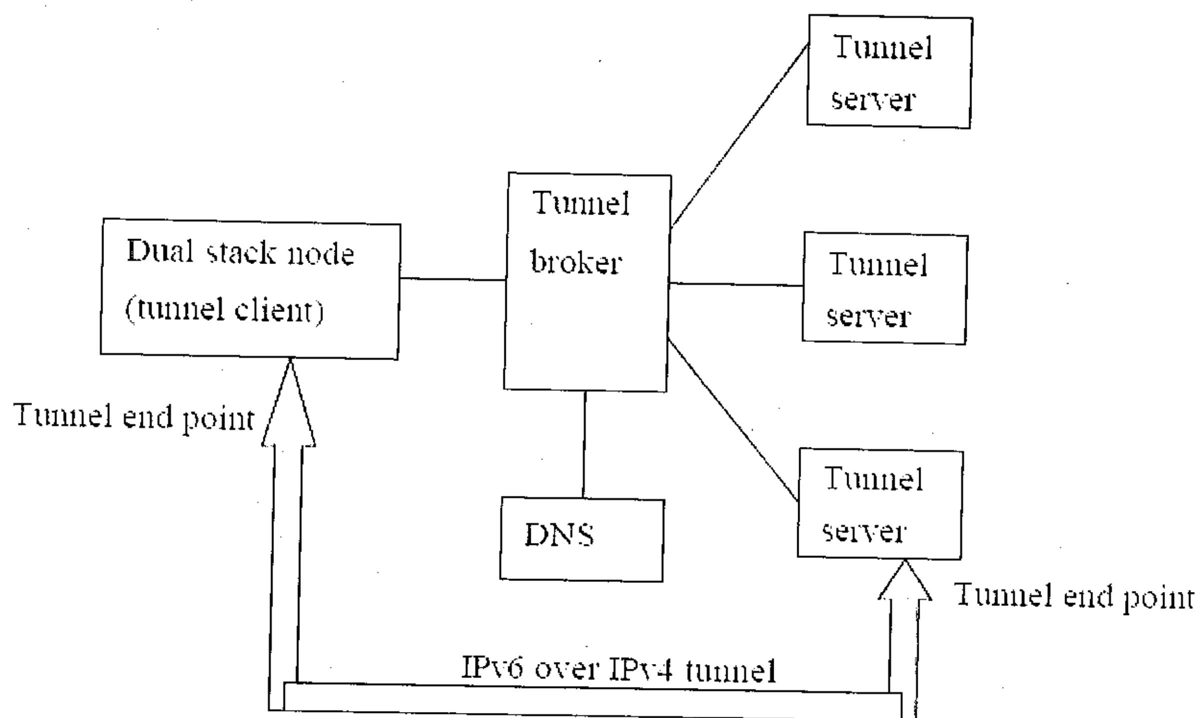


Figure 15 : The architecture of Tunnel Broker

TSP: Tunnel Setup Protocol

TSP is Tunnel Setup Protocol. It is a general method designed to simplify the setup of IPv6 over IPv4 tunnels. Tunnel broker is an implementation of tunnel setup protocol.

6over4

6over4 is a tunneling mechanism which allows isolated IPv6 hosts, located on a physical link which has no directly connected IPv6 router, to become fully functional IPv6 hosts by using an IPv4 multicast domain as their virtual local link. Thus, at least one IPv6 router using the same method must be connected to the same IPv4 domain their for IPv6 connectivity. 6over4 is not recommended.

ISATAP: Intra-Site Automatic Tunnel Addressing Protocol

ISATAP is Intra-Site Automatic Tunnel Addressing Protocol. It is the alternative to 6over4 and used inside a site. ISATAP automatically connects isolated IPv6 hosts or routers, which are called ISATAP nodes, within an IPv4 site via automatic IPv6- in-IPv4 tunnel, which uses the site's IPv4 infrastructure as an NBMA (Non-broadcast multi-access) link layer. It supports both address autoconfiguration and manual configuration. The IPv4 address of ISATAP link need not be globally unique.

64 bits	32 bits	32 bits
Link local, site local or global unicast	0000:5EFE	IPv4 address of ISATAP link

Figure 16 : IPv6 address format of ISATAP node

DSTM: Dual Stack Transition Mechanism

DSTM is the Dual Stack Transition Mechanism. Different from the previous tunneling mechanisms which are based on IPv6-over-IPv4 tunnels, DSTM is based on IPv4-over-IPv6 tunnels to carry IPv4 traffic within an IPv6-only network, and also provides a method to allocate a temporary IPv4 Address to IPv6/IPv4 nodes. DSTM can be used in a situation where the network infrastruc ture only supports IPv6, but some of the hosts on the network have dual-stack capability and IPv4 only applications.

DSTM consists of three components:

- DSTM server, which maintains temporary IPv4 addresses pool
- DSTM gateway, which is responsible for encapsulating and decapsulating IPv4packets over IPv6 packets.
- A dual stack host, which is called a DSTM node and wants to communicate using IPv4.

The working procedure is:

1. Dual Stack host, which wants to communicate using IPv4, requests a temporary IPv4 address from DSTM server
2. DSTM server reverses an IPv4 address for dual stack host from the IPv4 address pool and sends a reply to dual stack host. DHCPv6, TSP, RPC can be used to perform the task
   The following information is included in the reply:
   a. The allocated IPv4 address
   b. The period over which the address is allocated
   c. IPv4 and IPv6 addresses of the tunnel end point
3. The dual stack host configures the IPv4-over-IPv6 tunnel towards DSTM gateway. An IPv4-over-IPv6 tunnel is then set up between dual stack host and DSTM gateway. From now on, the IPv4 packets from dual stack host will be tunneled to DSTM gateway. DSTM gateway then decapsulates the packets from dual stack host and sends them to IPv4 internet, or encapsulates the packets from internet and sends them to dual stack host.

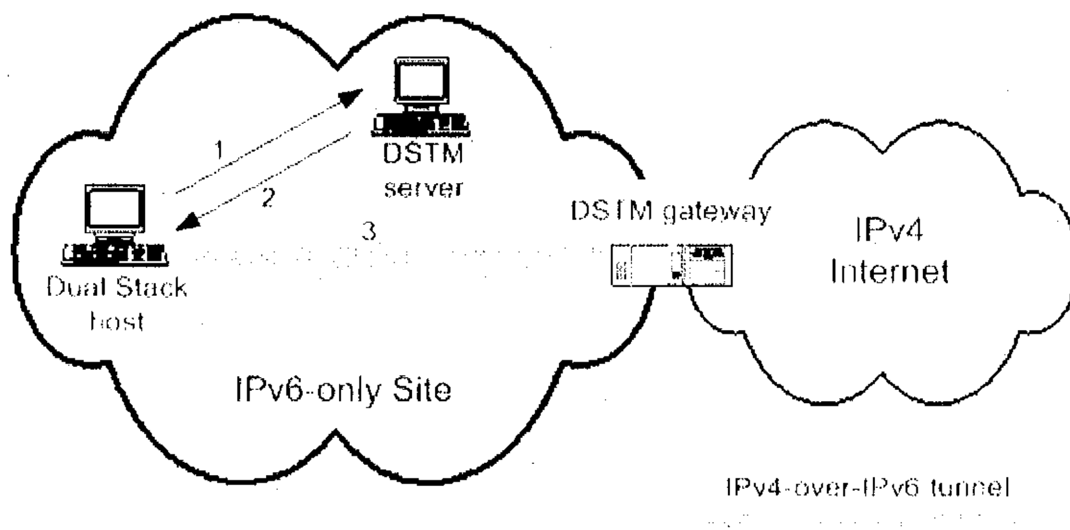


Figure 17 : DSTM Architecture and working procedure

Teredo

Shipworm or Teredo is a technique for the transport of UDP packets across NATs which works well in the scenario when there is a private IPv6 network behind the NAT machine. Teredo servers and relays are designated machines which allow the overlay of the Teredo network over the existing IPv4 network. IPv6 packets are encapsulated as UDP payload and are relayed by the Teredo relay which is available within the local network to the connected Teredo server; from there it is routed to the appropriate Teredo server nearest to the ultimate destination where the decapsulation of IPv6 is handled by the Teredo relay. Teredo is defined as a last resort mechanism to be used where 6 to 4 or other tunneling mechanisms are unavailable since there is an overhead due to the encapsulation into UDP.

BGP-tunnel

BGP tunnel explains how to interconnect IPv6 islands over an IPv4 cloud, including the exchange of IPv6 reachability information, using BGP.

1.2.4 Translation

When IPv6 islands are installed and connected together using one or several of the previous mechanisms, communication between IPv6 hosts is enabled. Communication between an IPv4 host and an IPv6 host may also need to be established. Since IPv4 and IPv6 hosts use different internet protocol, translation methods are deployed where IPv6-only devices wish to communicate with IPv4-only devices, or vice-versa. For the purpose of translation, translator must locate between IPv4 network and IPv6 network. Figure 16 illustrates the basic function and location of translator.

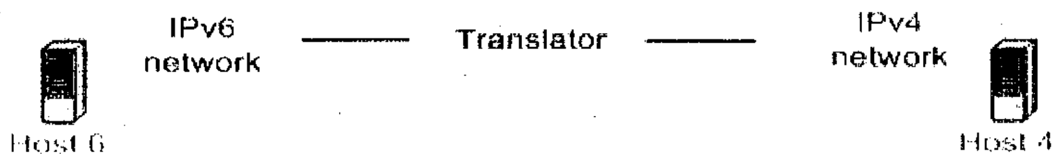


Figure 18 : Translation between IPv4 network and IPv6 network

SIIT: Stateless IP/ICMP Translation Algorithm

SIIT is Stateless IP/ICMP Translation Algorithm. SIIT describes a method to translate between IPv6 packet header and IPv4 packet header. The translation is limited to the IP packet header, and does not describe a method to assign a temporary IPv4 address to the IPv6 node. The translator is operating in a stateless mode, which means that translation needs to be done for every packet. It can only translate semantics shared between IPv4 and IPv6 protocols. It uses an IPv4-mapped IPv6 address (formatted as ::FFFF:a.b.c.d, a.b.c.d is IPv4 address) to describe the destination that is not IPv6 capable.

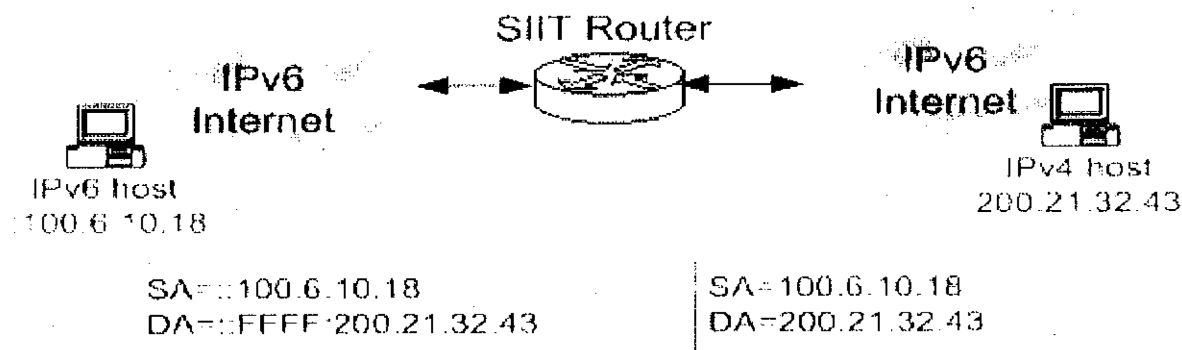


Figure 19 : Stateless IP/ICMP Translation Algorithm

NAT-PT: Network Address Translation with Protocol Translation

NAT-PT is Network Address Translation with Protocol Translation, which is a service that can

be used to translate an IPv4 packet into a semantically equivalent IPv6 packet and vice versa. It provides a combination of IPv4/IPv6 address translation and IP4/IPv6 protocol translation. By installing NAT-PT between an IPv6 network and IPv4 network, all IPv4 users are given access to the IPv6 network without host modification; equally, all IPv6 users are given access to the IPv4 network without host modification. Because some applications, such as DNS and FTP, carry network addresses in payloads, and NAT-PT does not snoop the payload, NAT-PT needs to cooperate with application layer gateway (ALG) to realize the translation.

Application layer gateway is a mechanism to allow hosts behind firewalls or NAT gateways to use applications that would otherwise not be allowed to traverse the firewall or NAT gateway.

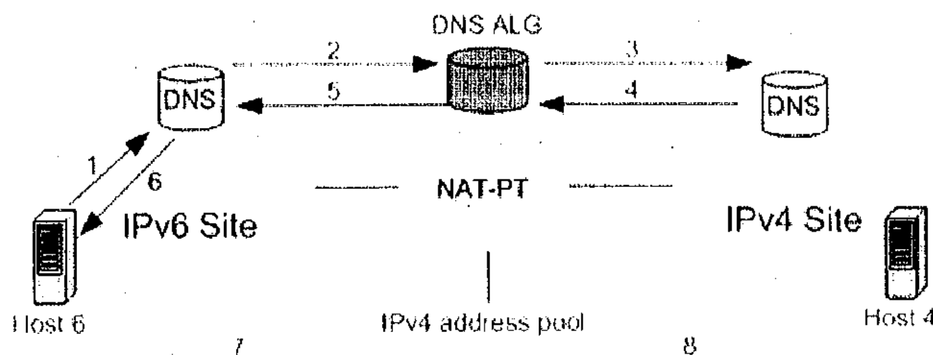


Figure 20 : Operation of NAT-PT

1. Host 6 (IPv6 host) sends an IPv6 DNS query, asking for the IPv6 address of host 4 (IPv4 host).

2. Because the DNS in IPv6 site has no record of the IPv4 host, the request is forwarded to the DNS-ALG

3. DNS-ALG translates the IPv6 DNS query into IPv4 DNS query, then sends it to the DNS in IPv4 site

4. The DNS in IPv4 site replies the IPv4 address of host 4. The reply is sent to DNSALG

5. DNS-ALG translates the IPv4 DNS reply to IPv6 DNS reply, in which the IPv4 address of host 4 is transformed to an IPv6 address containing prefix assigned to NAT-PT and IPv4 address of host 4 , then sends the IPv6 address to the DNS in IPv6 site

6. DNS in IPv6 site then sends the IPv6 DNS reply with the transformed IPv6 address of host 4 to host 6.

7. Host 6 sends IPv6 packets to host 4 through NAT-PT, using the transformed IPv6 address of

host 4 got from DNS in IPv6 site.

8. NAT-PT allocates an IPv4 address from its address pool to host 6. Then NAT-PT translates the IPv6 packets into IPv4 packets. The destination address for IPv4 packets will be the IPv4 address of destination Host 4, and the source address for the IPv4 packet is the one selected for Host 6 by NAT-PT. NAT-PT sends the translated IPv4 packets to host 4.

ALG: Application Layer Gateway

An Application Layer Gateway (ALG) is a mechanism to allow users behind firewalls or behind a NAT gateway to use applications that would otherwise not be allowed to traverse the firewall or NAT gateway.

In IPv6-only networks, the ALG functionality can be used to enable hosts in IPv6-only subnets to establish connections to services in the IPv4-only world and in some cases the other way around as well. This can be achieved by setting up ALGs on dual-stack hosts which have both IPv6 and IPv4 connectivity.

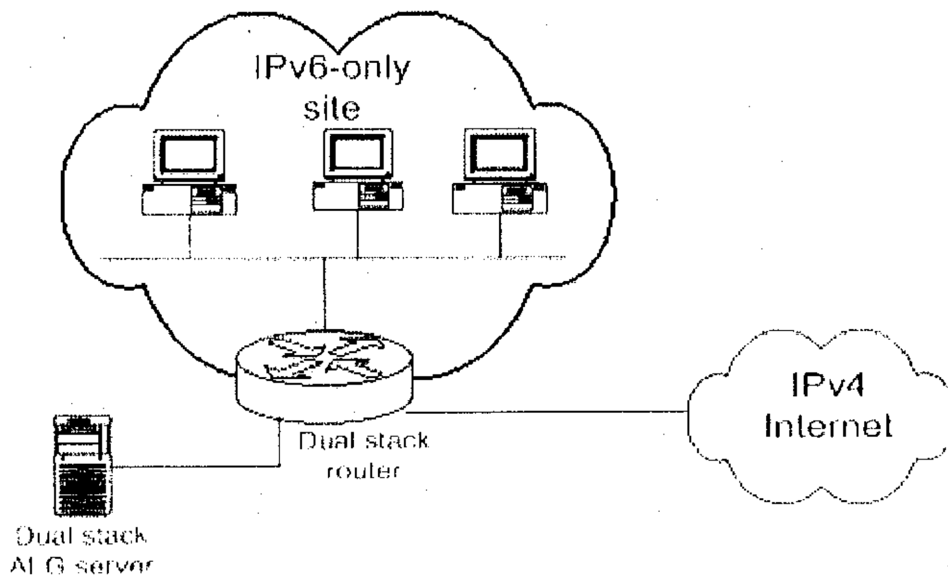


Figure 21 : Application Layer Gateway

TRT: Transport Relay Translator

TRT is Transport Relay Translator, which is a translator located in the transport layer. TRT enables IPv6-only hosts to communicate with IPv4-only hosts through translation of UDP or

repay of TCP.

We take client-server mode as example to explain this.

If the IPv6 client and the IPv4 server communicate using TCP, TRT locates between the the IPv6 client and the IPv4 server, terminates the IPv6 TCP session from the IPv6 client, and acts as an IPv6 server to the IPv6 client. At the same time, it originates a second IPv4 TCP session with the IPv4 server, and copies received data from each session to the other.

If the IPv6 client and the IPv4 server communicate using UDP, TRT locates between the IPv6 client and the IPv4 server, receives IPv6 UDP datagram from the IPv6 client, translates them into IPv4 UDP datagram, and sends them to the IPv4 server.

To implement TRT, there should be a dedicated router, which is dual stack, at a site to translate {UDP, TCP}/IPv6 to {UDP, TCP}/IPv4 and vice versa. Also, there should be a DNS server with DNS-ALG, which can map IPv4 addresses to IPv6 addresses. No modification is necessary for IPv6 hosts and IPv4 hosts. The TRT system can work with most of the common internet applications: HTTP, SMTP, SSH and etc.

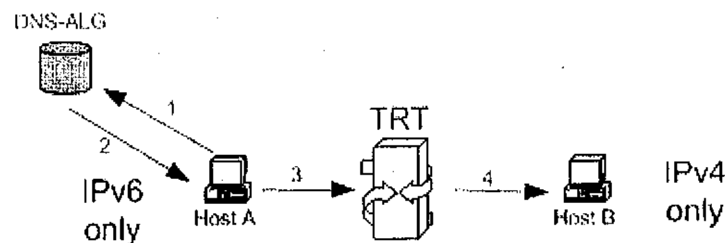


Figure 22 : Operation of TRT

1. Host A in IPv6 network sends a DNS query asking the IPv6 address of host B
2. DNS-ALG replies to host A with the IPv6 address of host B, which is constructed from the IPv4 address of host B and the special network prefix associated with the TRT.
3. The TRT then acts as reply or translator between host A and host B. Host A sends IPv6 packets to TRT.
4. TRT translates the IPv6 packets to IPv4 packets, and sends them to host B.

Sock64

Sock64 is mechanism that sock64 gateway accepts enhanced socks connections from IPv4 hosts and relays them to IPv4 or IPv6 hosts and vice verse, and thus enable IPv4 hosts to communicate

with IPv6 hosts. To use sock64 mechanism, the site must have socks aware client and a socks server. The mechanism does not request modification of the DNS system, because the DNS name resolving procedure at the client is delegated to the sock64 gateway.

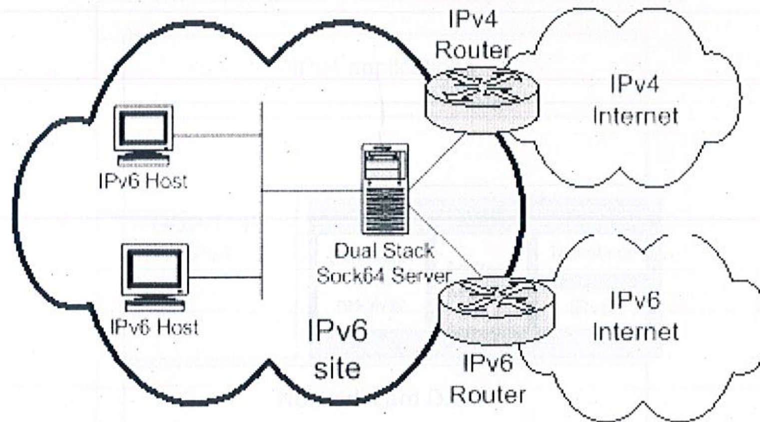


Figure 23 : Sock64 Architecture

BIS : Bump in the Stack

BIS is Bump in the Stack. It is a translation interface between IPv4 application and the underlying IPv6 network, and it assumes an underlying IPv6 infrastructure. BIS allows IPv4 applications running on an IPv4 host to communicate with IPv6 only hosts, or allows IPv6 hosts to communicate with other IPv6 hosts using existing IPv4 applications.

BIS mechanism consists of 3 components:

1. Translator: It receives IPv4 packets from the upper layer, translates the IPv4 packets into IPv6 packets and using the IP conversion mechanism defined in SIIT, and sends them to IPv6 network. Or receives IPv6 packets from IPv6 network, translates the IPv6 packets into IPv4 packets and sends them to the upper layer.

2. Extension name solver: It is responsible for returning a "proper" reply the IPv4 application's DNS request. If the IPv4 application sends a DNS request asking the IPv4 address of destination host, the extension name resolve will intercepts the request and sends a modified request asking the IPv4 and IPv6 address of the destination host. If IPv4 address is resolved, extension name solver will send the IPv4 address to IPv4 application; if only IPv6 address is resolved, it will request Address mapper to assign an IPv4 address and send the constructed IPv4 address to IPv4

application.

3. Address mapper: It is responsible for the assignment of IPv4 address, according to the request of translator or extension name server, and maintains the IPv4 address and IPv6 address table.

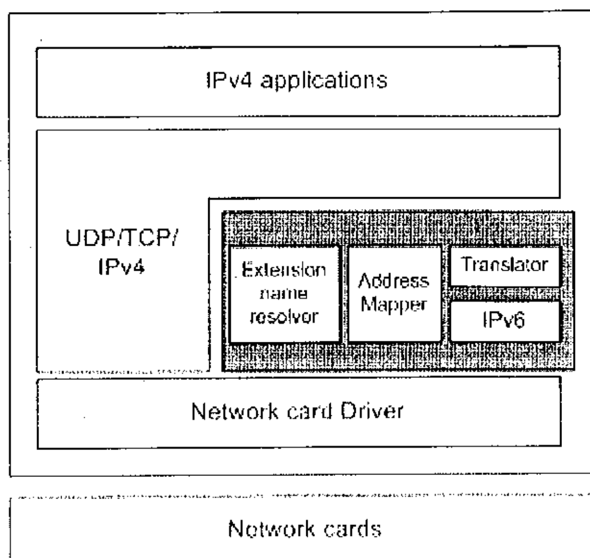


Figure 24 : Structure of dual stack host with BIS

BIA: Bump in the API

BIA is Bump- in-the-API. BIA inserts an API translator between the socket API and the TCP/IP module of the host stack. When IPv4 applications want to communicate with other IPv6 hosts, BIA detects the socket API functions from IPv4 applications and invokes the IPv6 socket API functions to communicate with the IPv6 hosts, and vice versa. In order to support communication between IPv4 applications and the target IPv6 hosts, pooled IPv4 addresses will be assigned through the extension name resolver in the BIA. Using BIA, both TCP(UDP)/IPv4 and TCP(UDP)/IPv6 stacks are implemented on the hosts.

BIA consists of 3 components:

1. Extension name resolver: It is responsible for returning a "proper" reply the IPv4 application's DNS request. If the IPv4 application sends a DNS request asking the IPv4 address of destination host, the extension name resolve will intercepts the request and sends a modified request asking the IPv4 and IPv6 address of the destination host. If IPv4 address is resolved, extension name solver will send the IPv4 address to IPv4 application; if only IPv6 address is resolved, it will request Address mapper to assign an IPv4 address and send the constructed IPv4 address to IPv4 application.

2. Function mapper: It is responsible for translating IPv4 socket API functions into IPv6 socket API functions, and vice versa.

3. Address mapper: It is responsible for the assignment of IPv4 address, according to the request of translator or extension name server, and maintains the IPv4 address and IPv6 address table.

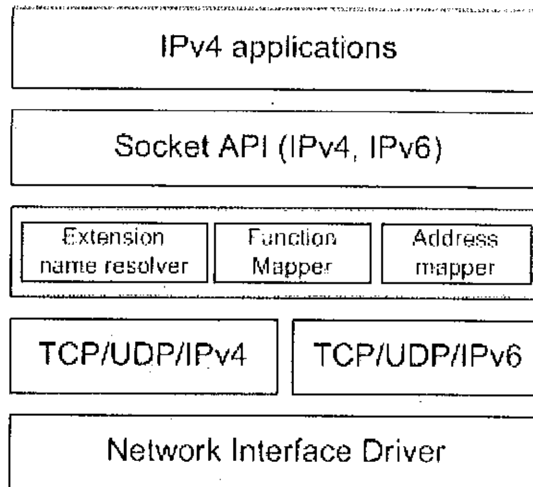


Figure 25 : Structure of Dual Stack host with BIA

1.3 SELECTION OF TRANSITION TECHNIQUE

The basic selection comes down to two broad choices- as DUAL STACK router configuration or TUNNELING.

If presented with a scenario where the router was open to configuration and upgrade, Dual Stack provides an obvious advantage. With router configuration the transition is seamless and transparent without any other changes to the existing network infrastructure.

Tunneling is used when changes in the existing network can only be software based and not hardware (router) changes.

We have used "Configured tunneling with payload transfer" to test our transition scheme. The configured tunnel application handles packets on a specific address and port and suitably strips data from IPv4/6 to IPv6/4 format.

In Payload Transfer tunneling mechanism we simply strip the data from an IPv4 packet and enclose it in an IPv6 packet if the packet is sent from an IPv4 network to an IPv6 network. Likewise, we enclose an IPv6 packet's data in an IPv4 header if the packet is sent to an IPv4 network from an IPv6 network.

2. DESIGN OF APPLICATION

2.1 Statement Of Purpose

The application enables data transfer between IPv4 and IPv6 nodes only and handles the interoperability issue. It acts as a bridge between the two systems implementing different protocols that is either IPv4 or IPv6. A server and a client is also formulated which transfer files from the latter to the former in different IP protocols.

2.2 Context of the Application

Environment: User who acts as a controller and inputs the required inputs for the application to work. The application responds to the user by showing a status report. The other external terminators are a host machine and a server machine implementing different protocols and interact with the application by sending and receiving data in the packets supported by respective protocols.

2.3 DFD(Data Flow Diagram)

The following data flow diagrams depicts the flow of data across implemented module. The terminators interacting with the application are user and host/server. Initially the user requests for an application instance by entering required parameters, the verify parameters modules verifies the input data format and all the required inputs if there is any inconsistency the error information is passed to the status window which is displayed to the user by display module. The verified data is then passed to the event handler which links the GUI to initialize module by invoking it and passing the parameters with required input. The initialize modules initializes the variables and sockets necessary for communication. Both modules reports any error message to display module. Main modules start execution with a reference to all the initialized variables and sockets. It mainly invokes the three modules InitializeBtoSthread, BtoSthread, HtoBthread which are the parallel threads of execution. Btosthread interact with the server to send the data while HtoBthread interacts with host side to receive the data sending. All the protocol conversions are done in the respective modules. The main thread servers one more purpose that is purging old threads who have finished execution. All these modules reports to display module in case of any error occurs.

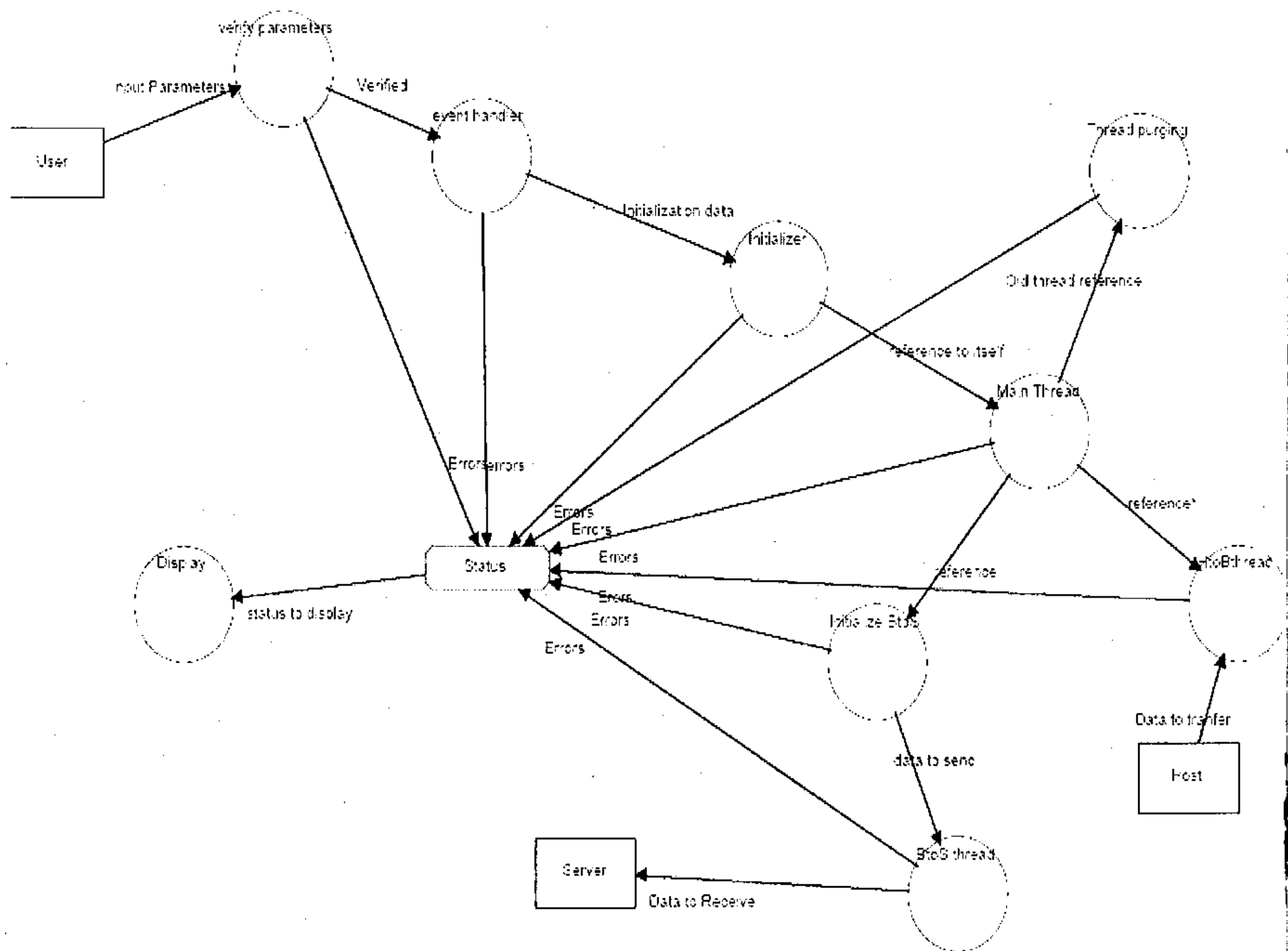


Figure 26 : Data Flow Diagram for tunnel application

2.4 Class and Structure Specification

Class Name: CNetworkInterface

Responsibilty:When the input parameters are received from the GUI interface they are stored in the member variable NIParameters. The member functions of this class performs the tunneling of data from the client to the server. The member functions create sockets for the both ends and the data is converted from IPv4 to IPv6 and vice versa. 3 threads are created one to handle the data from client to server other from server to client and another the handle the main application.

Members Functions Prototype:

Public:

int Initialize();

int InitializeBtoSThread(struct TwinSockets *mySocketsChain);

int Stop();

CNetworkInterface(struct Parameters *myParameters=NULL);

virtual ~CNetworkInterface();

private:

static UINT MainThreadLoop(LPVOID pthis);

static UINT HtoBThreadLoop(LPVOID pthis);

static UINT BtoSThreadLoop(LPVOID pthis);

Member Variables:

private:

CWinThread* MainThread;

CWinThread* HtoBThread;

CWinThread* BtoSThread;

SOCKET HtoBSock;

struct TwinSockets *SocketsChain;

struct Parameters *NIParameters;

Member Functions Specification:

CNetworkInterface():constructor initializes the variables with *input* parameters.The structure *Parameters* is used to *initialize* all variables.

~CNetworkInterface():Default destructor which acts as the garbage collector once the application is exited.

Initialize():This function is used to create a socket which will listen to incoming connections on the specified port. It also creates a main thread which is executed by the mainthreadloop() function. The function returns 0 on proper execution nor a non-zero number.

MainThreadLoop(): It is declared in the manner specified as this is the format a thread handle function has to be declared. The purpose of this function is to create a new twinsockets object which handles the later functionality. If the user stops and starts the application then a linked list is formed because of this function and the earlier nodes of the linked list is purged in this function. Out here InitializeBtoSThread() function is called and the thread from HtoB(host to application) is created and the thread from BtoS(Application to server) is created.

InitializeBtoSThread(): This function is used to create the connection between the application and the server. The socket is created and connected to the specified address and port of the server. The local name of the server and the address of the local socket is also retrieved.

HtoBThreadLoop(): This is the handle function which is executed when the thread is created which handles the data flow from the client to the server. Out here the data is taken from the client accepting socket and send on the connecting socket of application to the server.

BtoSThreadLoop(): This is the handle to the function which handled the data flow from the server to the application if there is any. Out here as the data is received from a connecting socket it is blocking.

Stop(): This is a garbage collector function which terminates all threads and the linked list which is kept hanging.

Variables Specification:

MainThread: Pointer of type *CwinThread* which is the handle of the main network thread.

HtoBThread: Pointer of type *CwinThread* which is the Handle of the thread serving connections between Host and Application.

BtoSThread: Pointer of type *CwinThread* which is the Handle of the thread serving connections between Application and Server.

HtoBSock:Object of socket class.It handles the socket which is listening to connections on the specified port.

SocketsChain:Pointer to a TwinSockets type structure.

NIParameters:Pointer to a parameter type structure used for *initializing* all the class variables at once.

Structures used:

```
struct TwinSockets {
        CWinThread* HtoBThread;
        CWinThread* BtoSThread;
        int HtoBThreadEnded;
        int BtoSThreadEnded;
        int SocketType;
        SOCKET HtoBSock;
        SOCKET HtoBConnSocket;
        SOCKET BtoSConnSocket;
        int ThreadNumber;
        struct TwinSockets *next;
} TwinSockets;
struct Parameters {
        CString ReceivingInterface;
        CString SendingInterface;
        CString HtoBPort;
        CString ServerAddress;
        CString BtoSPort;
        int i4To6;
        int SocketType;
```

int MaxConn;

} Parameters;

Twinsockets: This structure is used for starting and purging all the threads with the start and end of application.Every time the application is restarted a new node to the Twinsockets list is added, The current node is deleted from the list

and all the input parameters are passed in the new node which initializes the application again and the main thread starts to execute.

Parameters: This structure is used mainly to take all the inputs specified in the GUI of the application.

2.5 Flow Charts

Constructor of the CNetworkInterface Class

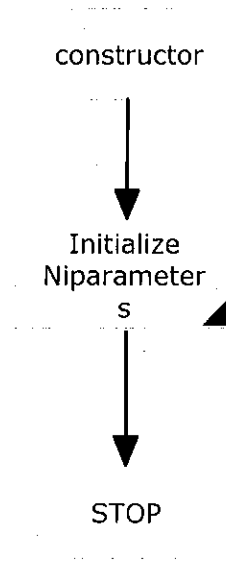


Figure 27 : Flow chart of the constructor function of the class CNetWorkInterface

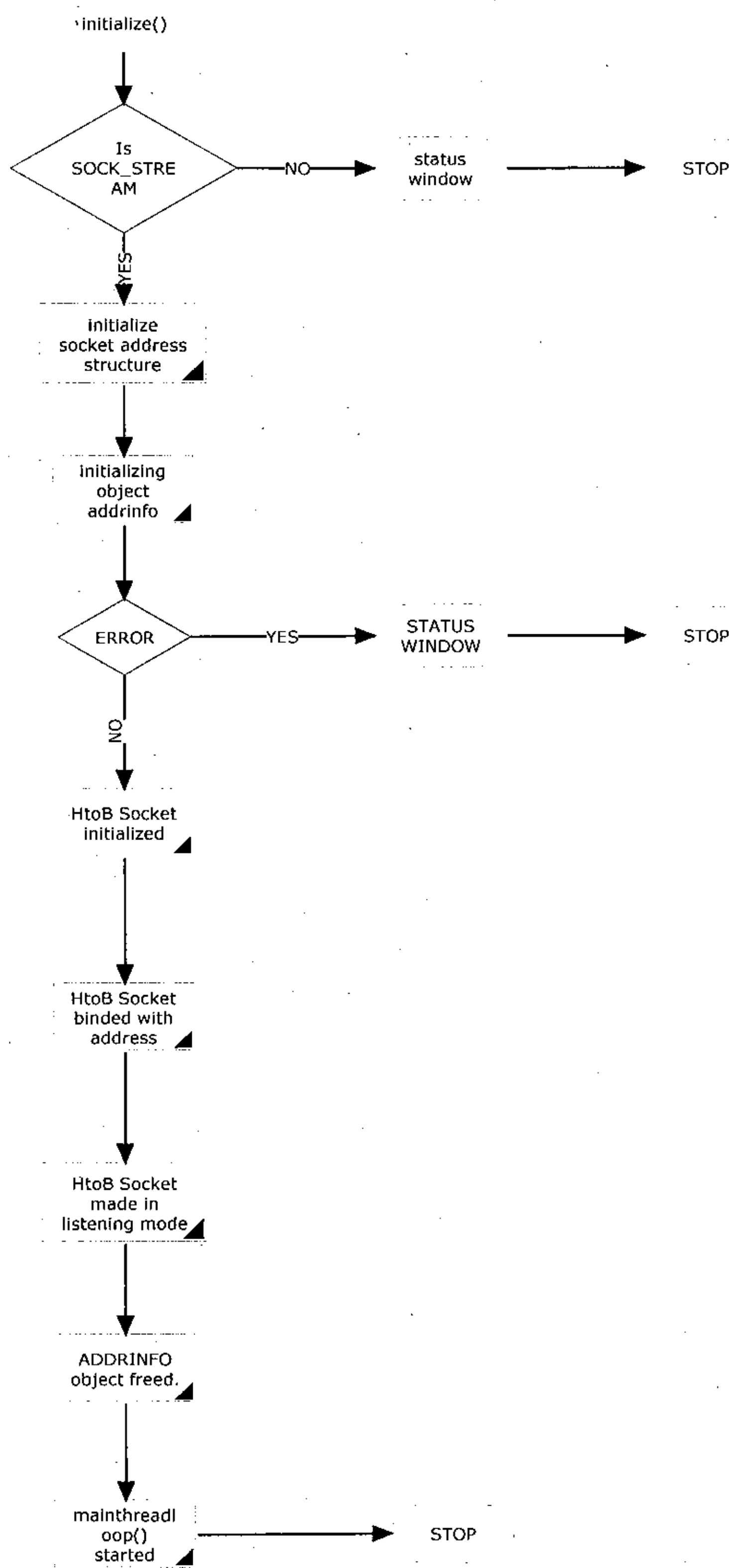


Figure 28 : Initialize Function of CNetworkInterface

Function MainThreadLoop()

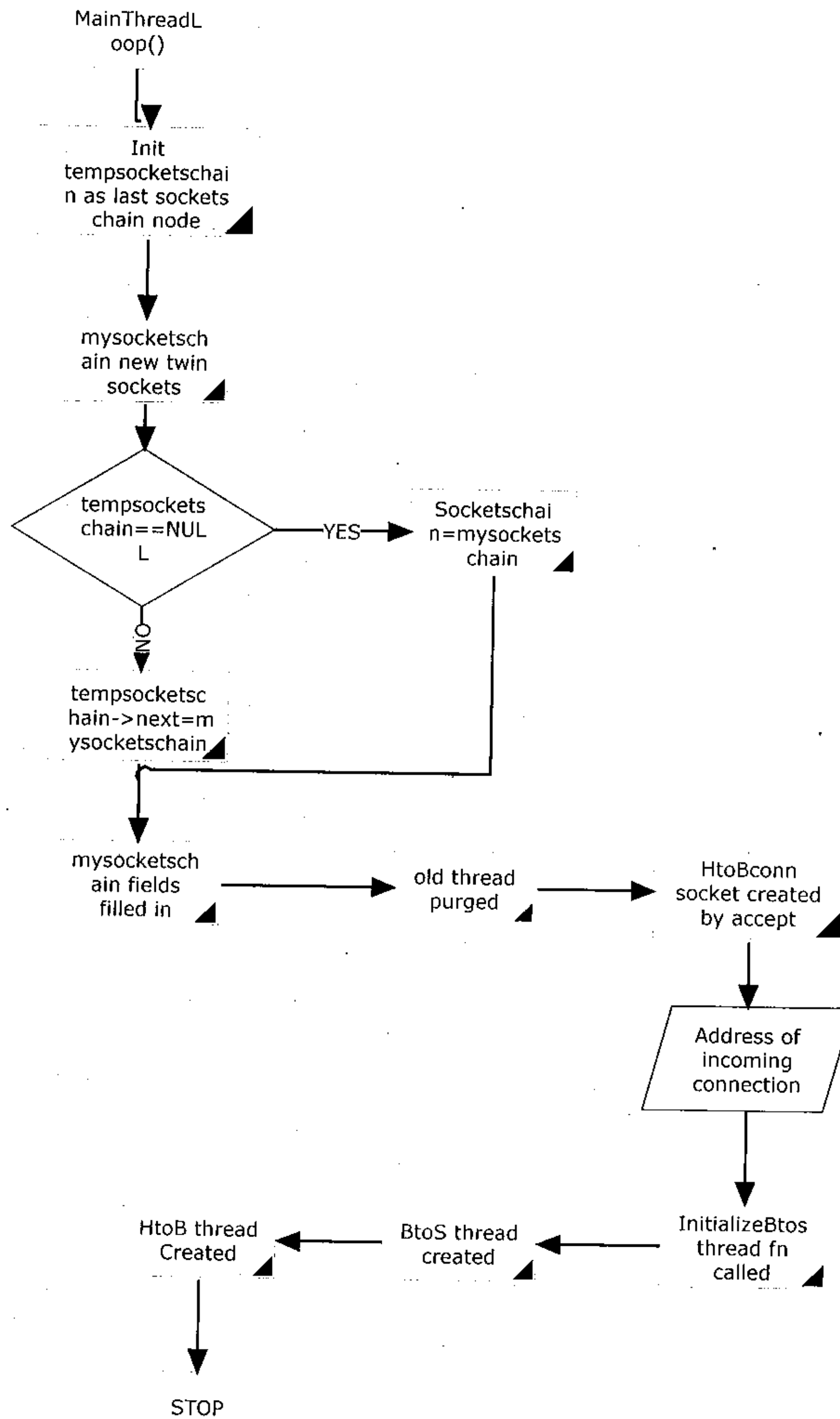


Figure 29 : Flow Chart of the MainThreadLoop() Function

Function InitializeBtoSThread()

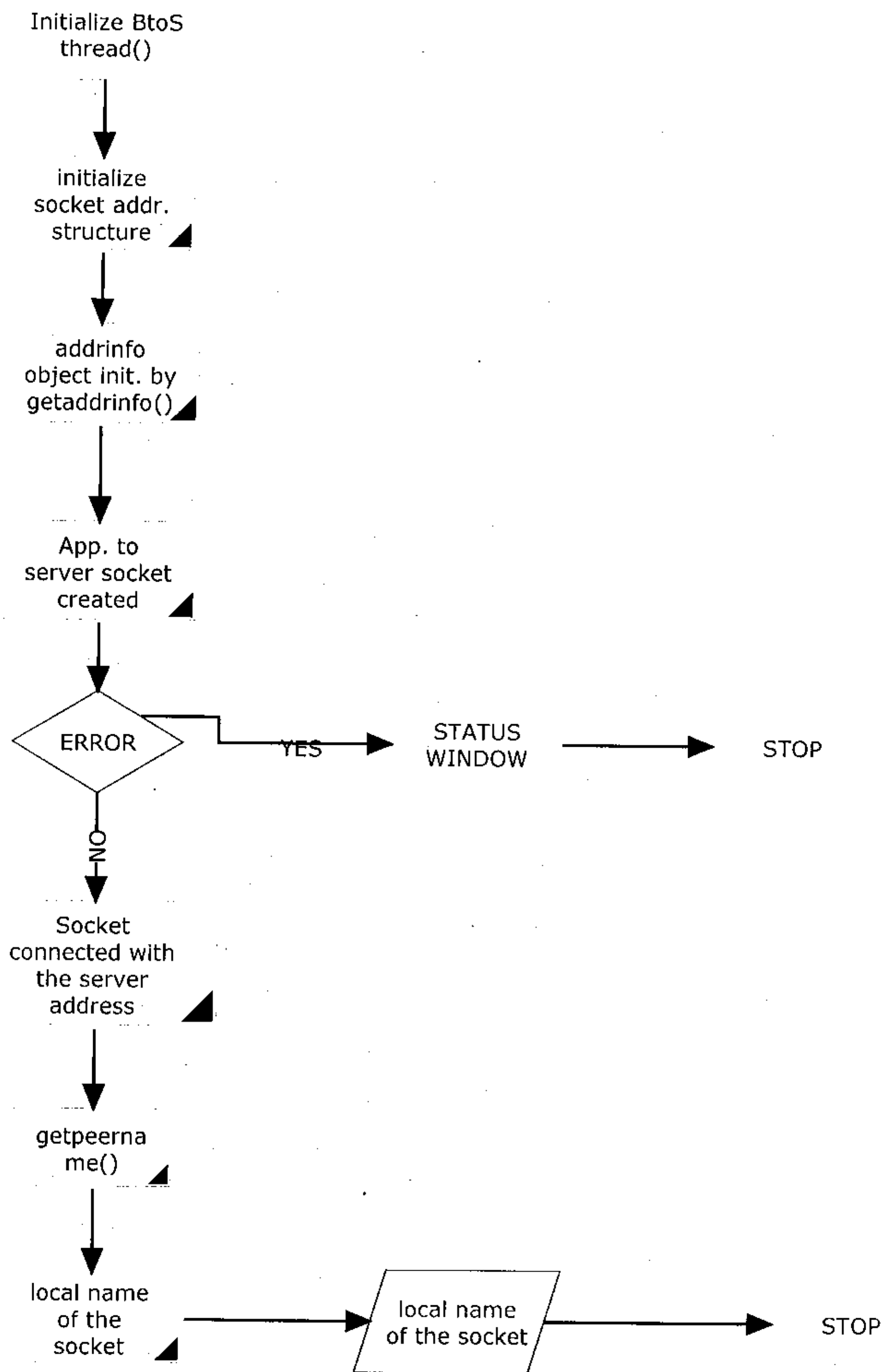


Figure 30 : Flow Chart of the function InitializeBtoSThread

Function BtoSthreadloop

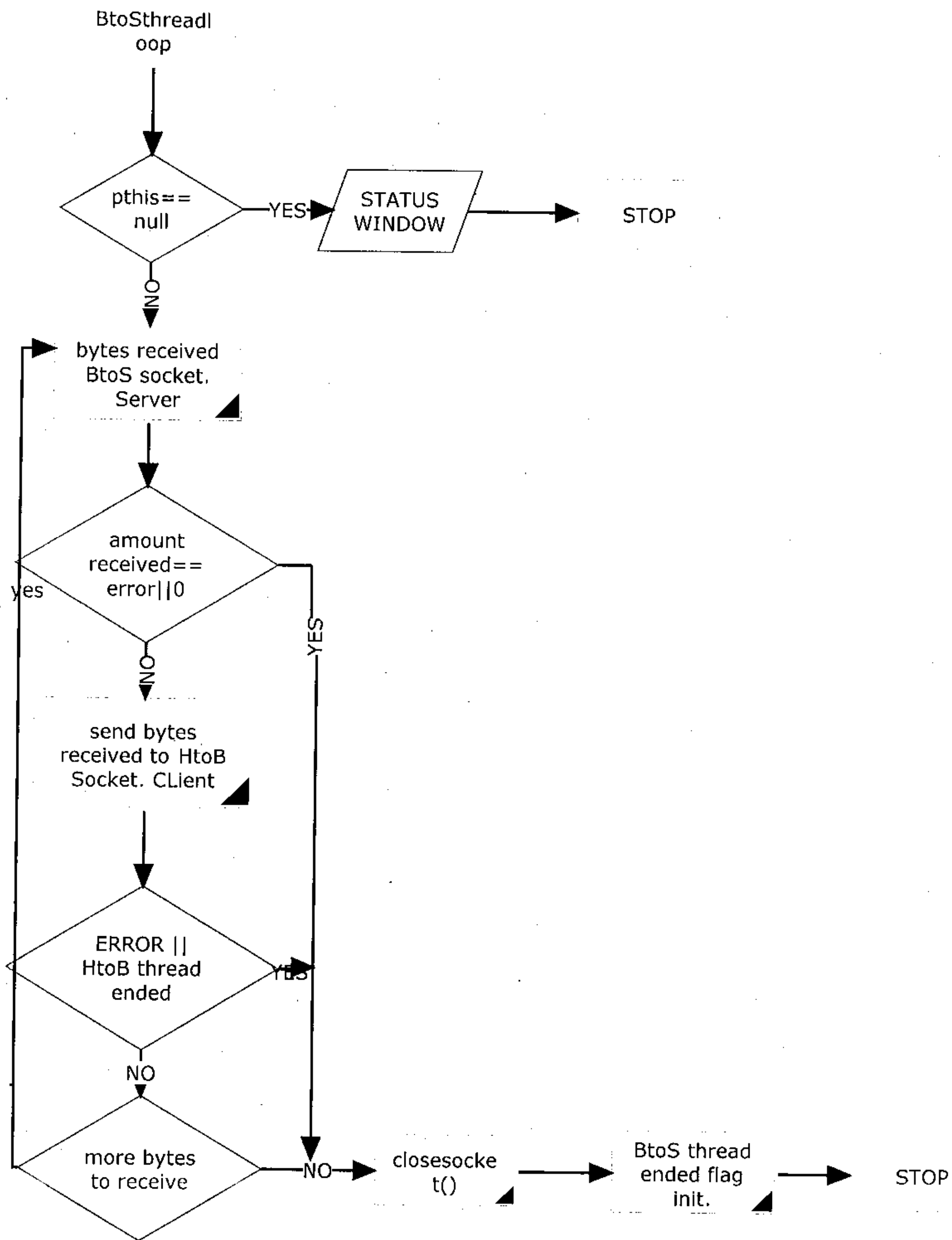


Figure 31 : Flow Chart of the function BtoSthreadLoop

Function HtoBthreadloop()

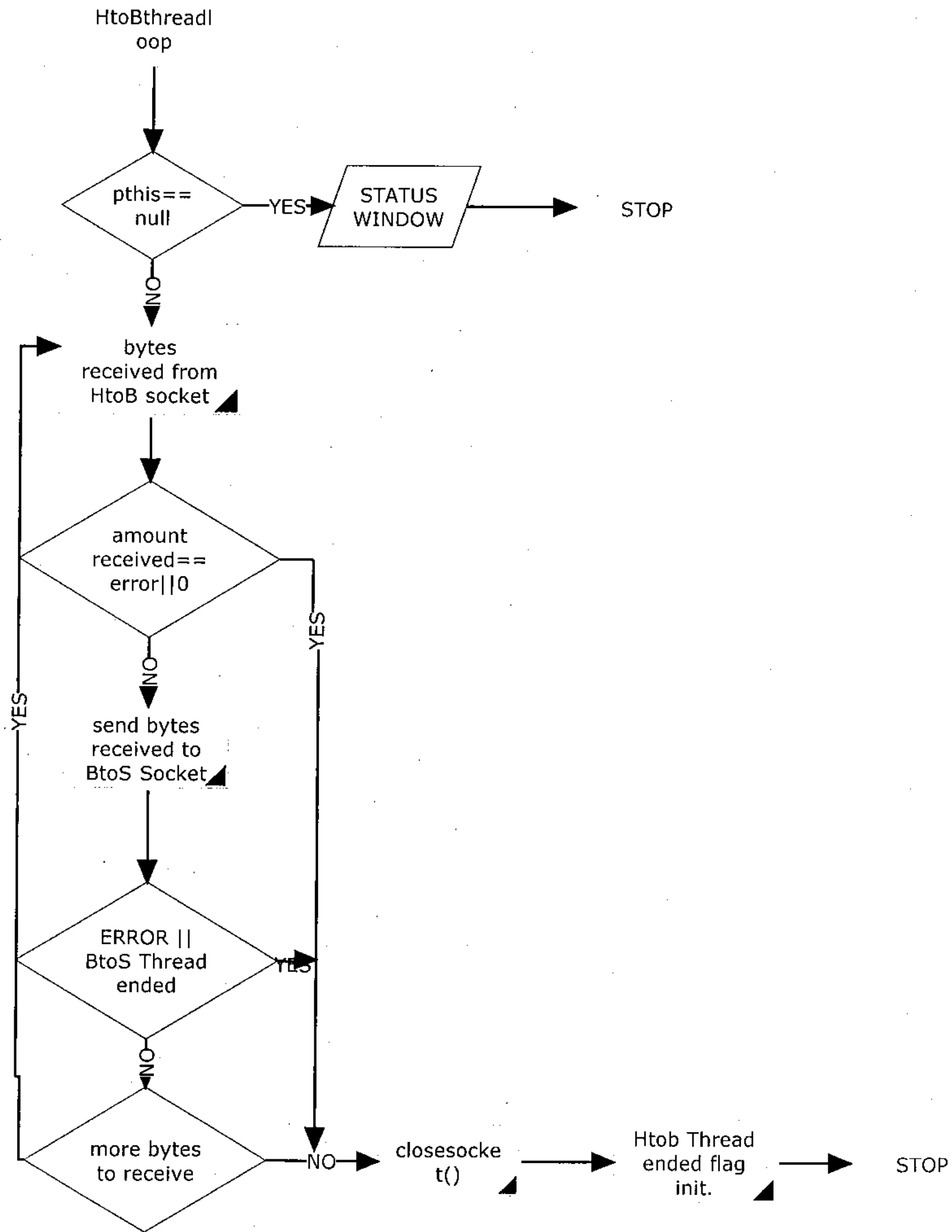


Figure 32 : Flow Chart of the function HtoBThreadLoop()

Function stop()

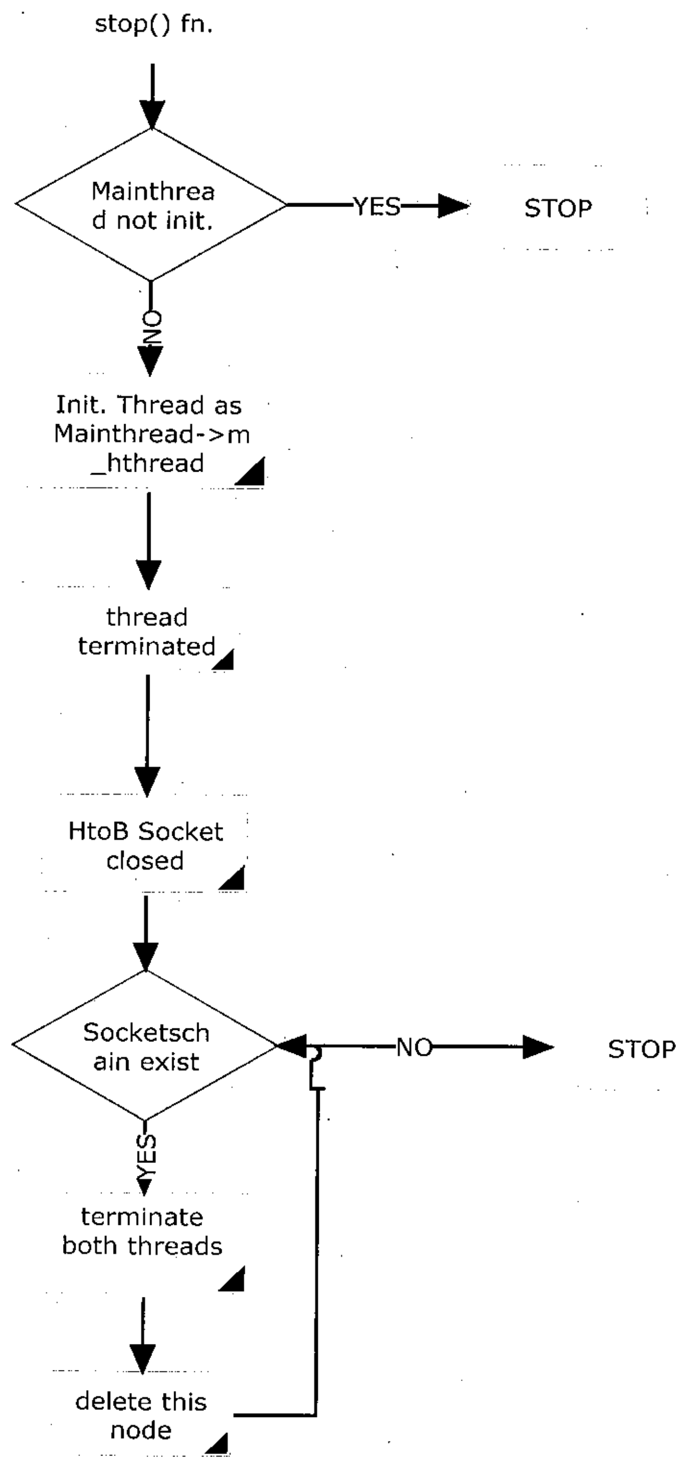


Figure 33 : Flow Chart of the function Stop()

2.6 Test Application

2.6.1 Class and Structure Specification

Class Name: Wcomm

Responsibilty: This class is used to initiate a instance of a server and a client. The server and the client can be of IPv6/IPv4. This application is used to send files from the client in one protocol and receive the file on the server in another protocol.

Members Functions Prototype:

Public:

Wcomm(int i);

void ConnectServer(string ipadd, int port);

void startServer(int port,int y);

void waitForClient();

void closeconnection();

void fileSend(string fpath);

void fileRecieve();

Member Functions Specification:

Wcomm(int i): Constructor which creates a socket of IPv4 or IPv6 according to the choice indicated by the flag which is the variable i.

void connectServer(string ipadd,int port): This function is used by the client to connect to the server whose address is entered by the user at the console. The port of the server is pre-written in the program as 27000.

void startServer(int port, int y): This function is used by the server to make the socket which is already created a listening socket which can accept connections of IPv6 or IPv4 as specified by the program.

void waitForClient(): This function calls the blocking protocol accept which enables the server to accept incoming connections. It also prints out the address and port of the client which is connected with the server.

void closeConnection(): This function destroys the socket which was created by the program.

void fileSend(string fpath): This function enables the client to send the file in the specified protocol. The file is denoted by the full path of it in the object fpath. The file is send in chunks 1024 bytes.

void fileRecieve(): This function enables the server to accept the file in the specified protocol. The server accepts the file in chunks of 1024 bytes. The file is stored in the place from where the executable is run.

Class Name: Tester

Responsibility: The class houses the function main() from which the program starts executing. The main function of this class is to invoke the various functions of the class wcomm.

2.6.2 Test Application Flow chart

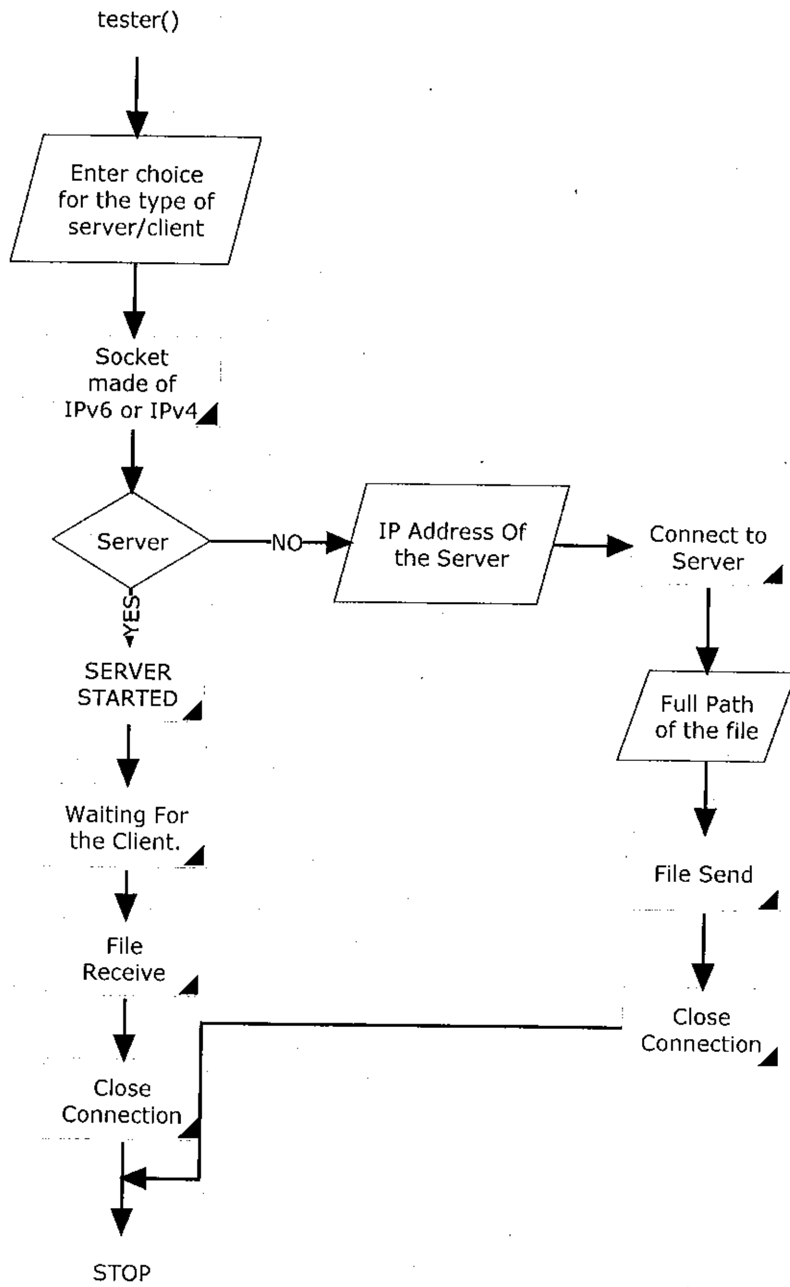


Figure 34 : Flow chart of the test application

3. IMPLEMENTATION

APPLICATION AND TEST APPLICATION SOURCE CODE

3.1 Application Source code

```
//NETWORK INTERFACE .cpp

//Including Self made header files.
#include "stdafx.h"
#include "46tunnel.h"
#include "NetworkInterface.h"
// dlgstatus is a object of class Cstatus which is predefined in MFC. It
is used to print
//the status messages.
extern CStatus dlgStatus;

// Code used to print the file name in the error slot. Self produced by
Visual Studio 2005
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif


// Constructor which takes a pointer variable of type parameters and
assigns it to
// the private member NIParameters.
CNetworkInterface::CNetworkInterface(struct Parameters *myParameters)
{
	NIParameters= myParameters;
}


//Destructor (optional) if NIParameters exist delete.
CNetworkInterface::~CNetworkInterface()
{
	if (NIParameters) delete NIParameters;
}


int CNetworkInterface::Initialize()
{
	// Addrinfo is a structure used to hold host information.
	ADDRINFO Hints, *AddrInfo;
	int RetVal;
	// Cstring is a predefined class in MFC.
	CString ctmp;
	SocketsChain= NULL;
	// Only TCP supported. So checking it.
	if  (NIParameters->SocketType != SOCK_STREAM) {
		dlgStatus.Print("CNetwork Interface: Socket type not
supported", 0);
		return -1;
	}
	/*memset is a function which initializes a structure with the second
argument. And the
	 third argument spcifies the size of the structure.*/
	memset(&Hints, 0, sizeof(Hints));
```

```
      /*the field of Hints which is an object of ADDRINFO is filled
according to the options
      selected. This object will enable the application to make a litening
socket.
      So if i4To6 is 1 then the family is IPV4 as the application has to
listen to IPv4 packets. */
      Hints.ai_family = (NIParameters->i4To6 == 1) ? PF_INET : PF_INET6;
    Hints.ai_socktype = NIParameters->SocketType;

    /* The AI_NUMERICHOST is initialized then the getaddrinfo attemps name
resolution.
      Setting the AI_PASSIVE flag indicates the caller intends to use the
returned
      socket address structure in a call to the bind function.*/
      Hints.ai_flags = AI_NUMERICHOST | AI_PASSIVE;

      //getaddrinfo provides protocol independent translation from host
name to address.
      //The parametrs passed are:
      /* NULL TERMINATED STRING CONTAINING THE HOST ADDRESS.
      PORT NUMBER.
      POINTER TO A ADDRINFO STRUCTURE THAT PROVIDES HINTS.
      POINTER TO A EMPTY ADDRINFO STRUCTURE CONTAINING RESPONSE OF THE
STRUCTURE.
      RETURN VALUE NON ZERO IF ERROR.*/
      RetVal = getaddrinfo( NULL,NIParameters->HtoBPort, &Hints,
&AddrInfo);

    //ERROR DETECTION
      if (RetVal != 0) {
            ctmp.Format("getaddrinfo failed with error %d: %s", RetVal,
gai_strerror(RetVal));
            dlgStatus.Print(ctmp, 0);
        return -1;
    }

      // This application only supports PF_INET and PF_INET6.
    if ((AddrInfo->ai_family != PF_INET) && (AddrInfo->ai_family !=
PF_INET6)) {
            dlgStatus.Print("CNetworkInterface: Socket type not
supported", 0);
            return -1;
      }

    /* The socket() function is used to create a new socket. The
parameters passed are:
      THE FAMILY OF THE CREATED SOCKET (PF_INET OR PF_INET6)
      THE TYPE OF SOCKET (SOCK_DGRAM || SOCK_STREAM || SOCK_RAW)
      THE TYPE OF PROTOCOL : IPV4 || IPV6
      THE RETURN TYPE IS A HANDLE TO THE NEWLY CREATED SOCKET. */
      HtoBSock = socket(AddrInfo->ai_family, AddrInfo->ai_socktype,
AddrInfo->ai_protocol);

      //Error Detection. WSAGetLastError() prints the last error occured
in words.
      if (HtoBSock == INVALID_SOCKET) {
```

```
        dlgStatus.Error("socket()", WSAGetLastError(), 0);
        return -1;
    }

    /*The bind() function associates a local address and port combination
with a new socket just created.
      This is most useful when the application is a server that has a
well-known port that clients
      know about in advance. The paremeters passed are:
      THE HANDLE OF THE SOCKET JUST CREATED.
      THE ADDRESS TO WHICH IT HAS TO BIND.
      THE LENGTH OF THE ADDRESS.
      RETURN TYPE IS OF TYPE SOCKET_ERROR IF THERE IS A ERROR.*/
    if (bind(HtoBSock, AddrInfo->ai_addr, AddrInfo->ai_addrlen) ==
SOCKET_ERROR) {
        dlgStatus.Error("bind()", WSAGetLastError(), 0);
        return -1;
    }

      /* the listen() function puts a socket in a state where it can
listen to incoming connections.
      The parameters passed are:
      THE SOCKET WHICH IS SUPPOSED TO LISTEN
      THE NO. OF MAX CONNECTIONS IT SHOULD ALLOW TO BE LISTENED.
      */
    if (listen(HtoBSock, NIParameters->MaxConn) == SOCKET_ERROR) {
            dlgStatus.Error("listen()", WSAGetLastError(), 0);
    return -1;
            }
    //Status message that the application is listening on the specified
port for the specified proctocol
      //for the specified number of connections.
    ctmp.Format("'Listening' on port %s, protocol %s, protocol family %s",
           NIParameters->HtoBPort, "TCP",(AddrInfo->ai_family == PF_INET)
? "PF_INET" : "PF_INET6");
      dlgStatus.Print(ctmp, 0);
      //The freeaddrinfo() function frees address information that the
getaddrinfo function
      //dynamically allocates in its addrinfo structures.
    freeaddrinfo(AddrInfo);
      /* The AfxBeginthread is used to create a new thread.
      The parameters are:
      The controlling Function for the working thread.
      The object of a class derived from CWinthread.
      The return value is a pointer to a newly created thread value. */
      MainThread = AfxBeginThread(MainThreadLoop, this);

      if (!MainThread) {
            dlgStatus.Print("Network Interface error: Error launching the
main thread", 0);
            return -1;
      }
      //The function initialize returns 0 upon properly execution.
      return 0;
}
```

```
int CNetworkInterface::InitializeBtoSThread(struct TwinSockets
*mySocketsChain)
{
      ADDRINFO Hints, *AddrInfo;
      char AddrName[1024];
      CString ctmp;
      //The sockaddr_storage structure stores protocol independent socket
address information.
      //sock_addr is specific to ipv4 and sock_addr6 to ipv6.
      struct sockaddr_storage Addr;
      int RetVal, AddrLen;

      /* By not setting the sockets flag to AI_PASSIVE we intend to say
that the returned address will be
      used for connect*/
    memset(&Hints, 0, sizeof(Hints));
    Hints.ai_family = (NIParameters->i4To6 == 1) ? PF_INET6 : PF_INET;
    Hints.ai_socktype = NIParameters->SocketType;
    RetVal = getaddrinfo(NIParameters->ServerAddress, NIParameters-
>BtoSPort, &Hints, &AddrInfo);

    if (RetVal != 0) {
        ctmp.Format("Cannot resolve address [%s] and port [%s], error %d:
%s",
                NIParameters->ServerAddress, NIParameters->BtoSPort,
RetVal, gai_strerror(RetVal));
            dlgStatus.Print(ctmp, mySocketsChain->ThreadNumber);
        return -1;
    }

      if (AddrInfo->ai_next != NULL) {
            dlgStatus.Print("More than one socket requested",
mySocketsChain->ThreadNumber);
        return -1;
      }

    // Creating the socket which connects the application to the receiving
server.
    mySocketsChain->BtoSConnSocket = socket(AddrInfo->ai_family, AddrInfo-
>ai_socktype, AddrInfo->ai_protocol);
    if (mySocketsChain->BtoSConnSocket == INVALID_SOCKET) {
        dlgStatus.Error("socket()", WSAGetLastError(), mySocketsChain-
>ThreadNumber);
        return -1;
    }

    //status message of attempting to connect to a specified server
      ctmp.Format("Attempting to connect to: %s",
            NIParameters->ServerAddress.GetBuffer(0) ? NIParameters-
>ServerAddress.GetBuffer(0) : "localhost");
      dlgStatus.Print(ctmp, mySocketsChain->ThreadNumber);
      //The connect() function establishes a connection with the specified
address to a specified socket.
      /* The parameters passed are:
      THE SOCKET ON WHICH THE CONNECTON HAS TO BE ESTABLISHED
      THE ADDRESS TO WHICH THE CONNECTION AS TO BE ESTABLISHED
      THE LENGTH OF THE ADDRESS
```

```
    THE RETURN IS SOCK_ERROR IF THERE IS A ERROR */
    if (connect(mySocketsChain->BtoSConnSocket, AddrInfo->ai_addr,
AddrInfo->ai_addrlen) == SOCKET_ERROR) {
          ctmp.Format("connect() to %s", NIParameters->ServerAddress);
        dlgStatus.Error(ctmp.GetBuffer(0), WSAGetLastError(),
mySocketsChain->ThreadNumber);
        return -1;
            }

    AddrLen = sizeof(Addr);
    /* The function getpeername() retrieves the name of the peer to which
the socket is connected to.
      The parameters passed are:
      THE SOCKET WHICH IS CONNECTED
      THE ADDRINFO OBJECT WHICH WILL TAKE THE RETRIEVING VALUES
      THE SIZE OF THE OBJECT IN BYTES.
      THE RETURN VALUE IS ZERO IF NO ERROR.*/
      if (getpeername(mySocketsChain->BtoSConnSocket, (LPSOCKADDR)&Addr,
&AddrLen) == SOCKET_ERROR)
      {
        dlgStatus.Error("getpeername()", WSAGetLastError(),
mySocketsChain->ThreadNumber);
    }
      else
      {/*The getnameinfo() function provides name resolution from an
address to the host name.
             POINTER TO A SOCKET ADDRESS STRUCTURE
             LENGTH OF THE SOCKET ADDRESS STRUCTURE
             POINTER TO THE HOST NAME
             LENGTH OF THE POINTER
             POINTER TO THE SERVICE NAME ASSOCIATED WITH THE PORT NUMBER
             LENGTH OF THE POINTER OF THE SERVICE NAME
             FLAGS ASSOCIATED WITH THE SOCKET.
             RETURNS 0 IF ON SUCCESSFUL COMPLETION.*/
         if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName,
sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
             strcpy(AddrName, "<unknown>");

         ctmp.Format("Connected to %s, port %d, protocol %s, protocol
family %s",
               AddrName, ntohs(SS_PORT(&Addr)),
               (AddrInfo->ai_socktype == SOCK_STREAM) ? "TCP" : "UDP",
               (AddrInfo->ai_family == PF_INET) ? "PF_INET" : "PF_INET6");
             dlgStatus.Print(ctmp, mySocketsChain->ThreadNumber);
     }
     freeaddrinfo(AddrInfo);

     AddrLen = sizeof(Addr);
       /* The getsockname() function is used to retrieve the local name of
the socket or the local address and port
       the system picked up for us. The parameters passed are:
       THE SOCKET WHICH IS CONNECTED.
       THE ADDR_INFO OBJECT ON WHICH THE VALUES ARE RETRIEVED
       THE LENGTH OF THE OBJECT IN BYTES.*/
       if (getsockname(mySocketsChain->BtoSConnSocket, (LPSOCKADDR)&Addr,
&AddrLen) == SOCKET_ERROR)
       {
```

```
        dlgStatus.Error("getsockname()", WSAGetLastError(),
mySocketsChain->ThreadNumber);
    }
      else
      {
        if (getnameinfo((LPSOCKADDR)&Addr, AddrLen, AddrName,
sizeof(AddrName), NULL, 0, NI_NUMERICHOST) != 0)
            strcpy(AddrName, "<unknown>");
        ctmp.Format("Using local address %s, port %d\n", AddrName,
ntohs(SS_PORT(&Addr)));
            dlgStatus.Print(ctmp, mySocketsChain->ThreadNumber);
    }

      return 0;

}


int CNetworkInterface::Stop()
{
      if (!MainThread) return FALSE;
      HANDLE thread=MainThread->m_hThread;
      /*The TerminateThread() function is used to terminate the
thread.Parameters passed are:
      THE THREAD TO BE TERMINATED
      THE EXIT CODE OF THE THREAD.
      ON SUCCESSFUL THE THREAD RETURNS A NONZERO VALUE. */
      BOOL res=TerminateThread(thread,0);
      //The closesocket() function is used to close a socket.
      closesocket(HtoBSock);
      dlgStatus.Print("Closing main thread", 0);

      //if no connections have been opened, SocketChain is still NULL
      // and the other threads are still to be created
      if (!SocketsChain) return 1;

      struct TwinSockets *mySocketsChain= SocketsChain->next;

      while (mySocketsChain= SocketsChain->next) {
            thread= SocketsChain->BtoSThread;
            res=TerminateThread(thread,0);
            thread= SocketsChain->HtoBThread;
            res=TerminateThread(thread,0);

            delete SocketsChain;
            SocketsChain= mySocketsChain;
      }

      return res;       //return 1 if everything works correctly
}


UINT CNetworkInterface::MainThreadLoop(LPVOID pthis)  //UINT & LPVOID is
required by AfxBeginThread()
{
static int ithread;
```

```
/* fd_set is used to make a set of sockets which is used to variuos
purposes. */
fd_set SockSet;
SOCKADDR_STORAGE From;
int FromLen;
char Hostname[1024];
CString ctmp;
      CNetworkInterface *myClass= (CNetworkInterface*) pthis;
    if (myClass == NULL) return -1;      // illegal parameter

      //Initializes the sockets in the set to zero.
    FD_ZERO(&SockSet);

      if (myClass->NIParameters->SocketType != SOCK_STREAM) return 0;

    while (1) {

            //Creates a new TwinSockets struct and inserts it into the
SocketChain tail
            struct TwinSockets *mySocketsChain= myClass->SocketsChain;
            struct TwinSockets *tempSocketsChain= NULL;

            while (mySocketsChain)
            {
                  tempSocketsChain= mySocketsChain;
                  mySocketsChain= mySocketsChain->next;
            }

            mySocketsChain= new TwinSockets;

            //insert the new SocketTwin as the new tail
            if (tempSocketsChain == NULL) myClass->SocketsChain=
mySocketsChain;
            else tempSocketsChain->next= mySocketsChain;

            ithread++;
            //Fills the remaining SocketTwin fields
            mySocketsChain->SocketType= myClass->NIParameters->SocketType;
            mySocketsChain->HtoBSock= myClass->HtoBSock;
            mySocketsChain->next= NULL;
            mySocketsChain->ThreadNumber= ithread;
            mySocketsChain->HtoBThreadEnded= 0;
            mySocketsChain->BtoSThreadEnded= 0;

            // Old threads purging
            struct TwinSockets *purgeSocketsChain= myClass->SocketsChain;
            tempSocketsChain= NULL;

            while (purgeSocketsChain) {
                  if ( (purgeSocketsChain->HtoBThreadEnded) &&
 (purgeSocketsChain->BtoSThreadEnded) )
                  {
                  HANDLE thread;
                  BOOL res;

                        thread= purgeSocketsChain->BtoSThread;
                        res=TerminateThread(thread,0);
```

```
                    thread= purgeSocketsChain->HtoBThread;
                    res=TerminateThread(thread,0);

                    if (tempSocketsChain == NULL) {          // The
thread at the top of the list ended
                          myClass->SocketsChain= purgeSocketsChain-
>next;
                          delete purgeSocketsChain;
                          purgeSocketsChain= myClass->SocketsChain;
//otherwise next statement will give protection error
                    }
                    else
                    {
                          tempSocketsChain->next= purgeSocketsChain-
>next;
                          delete purgeSocketsChain;
                          purgeSocketsChain= tempSocketsChain->next;
//otherwise next statement will give protection error
                    }

              }

              tempSocketsChain= purgeSocketsChain;
              purgeSocketsChain= purgeSocketsChain->next;
        }
        // End old threads purging

        FromLen = sizeof(From);
        /*The accept() function permits an incoming connection attempt
on a socket.
        THE SOCKET WHICH IS IN THE LISTENING MODE.
        POINTER TO AN ADDR_INFO STRUCTURE WHICH RETRIEVES THE ADDRESS
OF THE INCOMING CONNECTION
        THE LENGTH OF THE OBJECT.
        RETURN OF THE TYPE SOCKET WHICH IS CONNECTED AND THE PREVIOUS
SOCKET IS KEPT ON LISTENING */
      mySocketsChain->HtoBConnSocket= accept(myClass->HtoBSock,
(LPSOCKADDR)&From, &FromLen);
      if (mySocketsChain->HtoBConnSocket == INVALID_SOCKET) {
              dlgStatus.Print("\n", mySocketsChain->ThreadNumber);
              dlgStatus.Error("accept()", WSAGetLastError(),
mySocketsChain->ThreadNumber);
        return -1;
      }

      if (getnameinfo((LPSOCKADDR)&From, FromLen, Hostname,
sizeof(Hostname), NULL, 0, NI_NUMERICHOST) != 0)
        strcpy(Hostname, "<unknown>");

        ctmp.Format("\nAccepted connection from %s", Hostname);
        dlgStatus.Print(ctmp, mySocketsChain->ThreadNumber);
        // The function to application to receiving server is called
out here.
        if (myClass->InitializeBtoSThread(mySocketsChain) != 0) return
-1;

        //The thread is formed which handle the application to server.
```

```
            mySocketsChain->BtoSThread= AfxBeginThread(BtoSThreadLoop,
mySocketsChain);

            if (!mySocketsChain->BtoSThread) {
                  dlgStatus.Print("Network Interface error: Error
launching the BtoS thread", mySocketsChain->ThreadNumber);
                  return -1;
            }

            //The thread is formed which handles the client to
application.
            mySocketsChain->HtoBThread = AfxBeginThread(HtoBThreadLoop,
mySocketsChain);

            if (!mySocketsChain->HtoBThread) {
                  dlgStatus.Print("Network Interface error: Error
launching the HtoB thread", mySocketsChain->ThreadNumber);
                  return -1;
            }
    }

      return 0;
}


UINT CNetworkInterface::HtoBThreadLoop(LPVOID pthis)
{
fd_set SockSet;
SOCKADDR_STORAGE From;
int FromLen, AmountRead, RetVal;
char Hostname[1024], Buffer[64000];
CString ctmp;
      TwinSockets *SocketTwin= (TwinSockets *) pthis;
    if (SocketTwin == NULL) return -1;    // illegal parameter
    FD_ZERO(&SockSet);
      while(1) {
            /*The recv() function recieves bytes of data from a connected
socket which has an incoming connection.
                  The parameters passed are:
                  THE HANDLE TO THE SOCKET WHICH HAS AN INCOMING
CONNECTION
                  THE BUFFER THE BYTES WILL BE READ
                  THE SIZE OF THE BUFFER.
                  The function returns the amount of bytes read. */
                  AmountRead = recv(SocketTwin->HtoBConnSocket, Buffer,
sizeof(Buffer), 0);
            if (AmountRead == SOCKET_ERROR) {
                        // client could send a reset in order to close its
connection
                        if (WSAGetLastError() != WSAECONNRESET )
                              dlgStatus.Error("HtoB recv()",
WSAGetLastError(), SocketTwin->ThreadNumber );
                        closesocket(SocketTwin->HtoBConnSocket);
                        break;
            }
            if (AmountRead == 0) {
```

```
                dlgStatus.Print("Client closed connection", SocketTwin-
>ThreadNumber);
                closesocket(SocketTwin->HtoBConnSocket);
                break;
            }
                /*The send() function sends a stream os bytes in a
connected socket. The parameters passed are:
                THE CONNECTED SOCKET.
                THE BUFFER WHICH CONTAINS THE BYTES TO BE SENT
                THE AMOUNT OF BYTES TO BE SENT.
                FLAGS OF HOW THE SENT HAS TO BE MADE. */
            RetVal = send(SocketTwin->BtoSConnSocket, Buffer, AmountRead,
0);
            if (RetVal == SOCKET_ERROR) {
                    if (SocketTwin->BtoSThreadEnded == 0)
                  dlgStatus.Error("HtoB send()", WSAGetLastError(),
SocketTwin->ThreadNumber);
                closesocket(SocketTwin->BtoSConnSocket);
                break;
            }
        }
            closesocket(SocketTwin->HtoBConnSocket);  //if not closed yet
            //Flag showing that the Client to application thread has been
ended.
            SocketTwin->HtoBThreadEnded= 1;
return 0;
}


UINT CNetworkInterface::BtoSThreadLoop(LPVOID pthis)
{
char Buffer[64000];
CString ctmp;
int RetVal, AmountRead;


      TwinSockets *SocketTwin= (TwinSockets *) pthis;
    if (SocketTwin == NULL) return -1;     // illegal parameter

      while (1) {

            memset(Buffer, 0, sizeof(Buffer));

            AmountRead = recv(SocketTwin->BtoSConnSocket, Buffer,
sizeof(Buffer), 0);
            if (AmountRead == SOCKET_ERROR) {
                  dlgStatus.Error("Is the server still alive? BtoS
recv()", WSAGetLastError(), SocketTwin->ThreadNumber);
                  closesocket(SocketTwin->BtoSConnSocket);
                  break;
            }
        if (AmountRead == 0) {
            dlgStatus.Print("Server closed connection", SocketTwin-
>ThreadNumber);
            closesocket(SocketTwin->BtoSConnSocket);
            break;
        }
```

```
        // Send the message.  Since we are using a blocking socket, this
        // call shouldn't return until it's able to send the entire
amount.
        RetVal = send(SocketTwin->HtoBConnSocket, Buffer, AmountRead, 0);
        if (RetVal == SOCKET_ERROR) {
                  if (SocketTwin->HtoBThreadEnded == 0)
                        dlgStatus.Error("Is the host still alive? BtoS
send()", WSAGetLastError(), SocketTwin->ThreadNumber );
            break;
        }
      }


      // SD_SEND: subsequent calls to the send function are disallowed.
      // For TCP sockets, a FIN will be sent after all data is sent and
      // acknowledged by the receiver.
    shutdown(SocketTwin->BtoSConnSocket, SD_SEND);

    closesocket(SocketTwin->BtoSConnSocket);
      SocketTwin->BtoSThreadEnded= 1;

      return 0;
}
```

```
//NETWORK INTERFACE. H

//Needed for the pre compiled header file
//self generated from Visual Studio 2005
#if
!defined(AFX_NETWORKINTERFACE_H__0B4E7D42_B872_43DB_8DD1_1A8BE842018F__INC
LUDED_)
#define
AFX_NETWORKINTERFACE_H__0B4E7D42_B872_43DB_8DD1_1A8BE842018F__INCLUDED_

#include <winsock2.h>
#include <ws2tcpip.h>


#if _MSC_VER > 1000
#pragma once
#endif

typedef struct TwinSockets {
	CWinThread* HtoBThread;		//	handle of the thread serving
connections between Host and Application
	CWinThread* BtoSThread;		//	handle of the thread serving
connections between Application and Server
	int HtoBThreadEnded;		//Flag of whether the thread have
ended or not.
	int BtoSThreadEnded;
	int SocketType;
	SOCKET HtoBSock;
	SOCKET HtoBConnSocket;
	SOCKET BtoSConnSocket;
	int ThreadNumber;
	struct TwinSockets *next;//Required for the linked list
} TwinSockets;

typedef struct Parameters {
	CString HtoBPort;		// Port the application is listening
on.
	CString ServerAddress;		// Server IPv4/IPv6 address
	CString BtoSPort;		// Server port: port whose traffic is
directed to
	int i4To6;			// IPv4 to IPv6 or vice versa
	int SocketType;			// TCP or UDP?
	int MaxConn;			// Maximum number of connections
waiting on the accept()
} Parameters;

class CNetworkInterface
{
public:
	int Initialize();
	int InitializeBtoSThread(struct TwinSockets *mySocketsChain);
	int Stop();
	CNetworkInterface(struct Parameters *myParameters= NULL);
	virtual ~CNetworkInterface();
```

```
private:    // Thread handlers
      static UINT MainThreadLoop(LPVOID pthis);
      static UINT HtoBThreadLoop(LPVOID pthis);
      static UINT BtoSThreadLoop(LPVOID pthis);

private:    // Variables
      CWinThread* MainThread;        //    handle of the main network
thread (blocked on the accept() call)
      CWinThread* HtoBThread;        //    handle of the thread serving
connections between Host and Application
      CWinThread* BtoSThread;        //    handle of the thread serving
connections between Application and Server
      SOCKET HtoBSock;
      struct TwinSockets *SocketsChain;
      struct Parameters *NIParameters;
};

#endif //
!defined(AFX_NETWORKINTERFACE_H__0B4E7D42_B872_43DB_8DD1_1A8BE842018F__INC
LUDED_)
```

3.2 Test Application Source Code

```
/* TEST APPLICATION SOURCE CODE
 * Following code implements a test application,which servers as an
IPv4/IPv6
   Client/Server depending on the choice which use makes.*/


//Namespaces Used:
using System;  //Gives access to all the access to all the classes and
methods that are directly under the System namespace
using System.Collections.Generic; //Namespace for using generics class
using System.Text;  //Namespace for using
using System.Net;
using System.Net.Sockets; //Namespace for using the socket class
using System.IO;  //Namespace for using Filing operations


//Wcomm class definition
public class Wcomm
{
    Socket msocket;
    Socket acceptSocket;

    /*default constructor where i is a flag for creating IPv6 or
      IPv4 socket used for starting the client or the server */

    public Wcomm(int i)
    {
    if (i==1 || i==4)
    {
        //creating an IPv6 socket,address family used is
InterNetworkV6,socket type is stream
        //for tcp connections and protocol used is tcp.
        msocket = new Socket(AddressFamily.InterNetworkV6,
                SocketType.Stream, ProtocolType.Tcp);
                Console.WriteLine("ipv6 server started"); //starting server
or client
    }
    else
    {
        //creating an IPv4 socket,address family used is
InterNetworkV4,socket type is stream
        //for tcp connections and protocol used is tcp.
        msocket = new Socket(AddressFamily.InterNetwork,
                SocketType.Stream, ProtocolType.Tcp);   //starting the
server or client.
    }

    }


    //public method used by client for connecting to the server.
    //ipaddress and the port is being passed as arguements.
```

```
    public void connectServer(string ipadd, int port)
    {
        IPAddress ip = IPAddress.Parse(ipadd); //string is being parsed to
a IPAddress type object
        IPEndPoint ipep = new IPEndPoint(ip, port); //An IPendPoint object
is created which keeps the ip address and the port no

        //Exception Handling while trying to connect
        try
        {
            msocket.Connect(ipep); //The connect method is called which
takes IPEndPoint object as arguement.
        }
        catch (SocketException e)
        {
            Console.WriteLine("Unable to connect to server."); //Message
to the user
            Console.WriteLine(e.ToString());
            return;
        }
    }

    //public method for starting the server
    public void startServer(int port,int y) //Port on which the server
listens and y is a flag to decide the protocol ie. IPv4 or IPv6
    {
        if (y == 1)
        {
            IPEndPoint ipep = new IPEndPoint(IPAddress.IPv6Any, port);
//IPAddress.IPv6Any has the IPv6 remote address.
            msocket.Bind(ipep); //bind() method is called on the socket.
        }
        else
        {
            IPEndPoint ipep = new IPEndPoint(IPAddress.Any, port);
//IPAddress.Any has the IPv4 remote address.
            //bind () Associates a Socket with a local endpoint.
            msocket.Bind(ipep); //bind() method is called on the socket
        }

        /* Listen causes a connection-oriented Socket to listen for
incoming connection attempts.
         * The backlog parameter specifies the number of incoming
connections that
         * can be queued for acceptance*/

        msocket.Listen(10); //the binded socket can listen up to 10
connections
    }
    //Waiting for the client
    public void waitForClient()
    {
        /* The accept method permits an incoming connection attempt on a
socket.
         * If no error occurs, accept returns a value of type SOCKET which
is a handle for the socket on
```

```
        * which the actual connection is made.
        */

        Socket client = msocket.Accept();
        IPEndPoint clientep = (IPEndPoint)client.RemoteEndPoint;
//Fetching the remote ip address and port info
        Console.WriteLine("Connected with {0} at port {1}",
clientep.Address, clientep.Port); //display
        msocket = client; //Handle of the conncetion socket is passed to
the original socket.
    }



    //Closing the connection
    public void closeConnection()
    {
      //The Close method closes the remote host connection and
      //releases all managed and unmanaged resources associated with the
Socket
        msocket.Close();
    }

    public void fileSend(string fpath) //path of the file is passed for
sending.
    {
        //Use the FileInfo class for operations such as copying, moving,
renaming,
        //creating, opening, deleting, and appending to files.


        FileInfo fi=new FileInfo(fpath);
        string filename=fi.Name;  //file name is being fetched from the
whole path
        Console.WriteLine(filename);
        byte[] fname = new byte[1024];
        fname=Encoding.ASCII.GetBytes(filename); //converting the filename
is byte stream
        //only byte stream is supported on send method


        int temp2=0;
        byte[] temp=new byte[1030]; //stream array used as buffer

        Stream inputStream=File.OpenRead(fpath); //the file is opened in
read only mode

        //while loop reads the whole file in chunks of 1024 bytes of data
         do
        {
            //temporary buffer for holding the data from file
            temp2=inputStream.Read(temp,0,1024); //reading 1024 bytes from
the file
            if (temp2==1024)
            {
                /*send() method Sends the specified number of bytes of
data to a
```

```
                * connected Socket, using the specified SocketFlags*/
                msocket.Send(temp,1024,SocketFlags.None); //sending the
data in buffer
            }
            else
            {
                msocket.Send(temp,temp2,SocketFlags.None); //sending the
data in buffer
            }
        }while(temp2==1024); //terminating the loop if data left is less
than 1024 bytes
        }

    //public method used by the server to receive the file.
    public void fileReceive()
    {
        byte [] buffer=new byte[1024]; //temporary buffer for holding the
data


        string filename=Encoding.ASCII.GetString(buffer);

        Stream we=File.Create("akhil.txt"); //creating a new file at the
server side to store the received data
        int x=0;
        do
        {
            //Receive mthod Receives data from a bound Socket into a
receive buffer.
            x=msocket.Receive(buffer);
            we.Write(buffer,0,x);//writing in the newly created file
        }while (x==1024);

    }
}

//Public tester class with the main function
public class tester
{
    static void Main(string[] args)
    {
        //choices for the application instance it can be a IPv4/v6
server/client

        Console.WriteLine("1. IPv6 Server Application");
        Console.WriteLine("2. IPv4 Client Application");
        Console.WriteLine("3. IPv4 Server Application");
        Console.WriteLine("4. IPv6 Client Application");
        string y = Console.ReadLine();
        int z = Convert.ToInt32(y); //coverting the input string choice in
to an integer
        Wcomm w = new Wcomm(z);

        //starting the server application
        if (z == 1 || z==3)
        {
            //RUNSERVER
```

```
            w.startServer(27015,z); //27015 is the default port for the
server and z is the flag
            Console.WriteLine("Server Started");
            if (true)
            {
               w.waitForClient(); //wating for client
               w.fileReceive(); //receiving the file
               w.closeConnection(); //closing the connections
            }
        }
        //starting the client application
        else
        {
            Console.WriteLine("Enter the ip address of server"); //address
of the server
            string q = Console.ReadLine();
            w.connectServer(q, 27000);    //connecting to the server of
entered ip address
            Console.WriteLine("connected to server");
            byte[] temp = new byte[1024];
            string x = "hELLO";
            x = Encoding.ASCII.GetString(temp);
            Console.WriteLine(x);
            Console.WriteLine("Enter the full path of the file");//path of
the file for sending
            string fpath = Console.ReadLine();
            w.fileSend(fpath); //calling the filesend method
            w.closeConnection(); //closing the connection from client side
        }
    }
}
```

4. EXPERIMENTAL SETUP AND TESTING

4.1 SETUP, TEST PROCEDURE & RESULTS

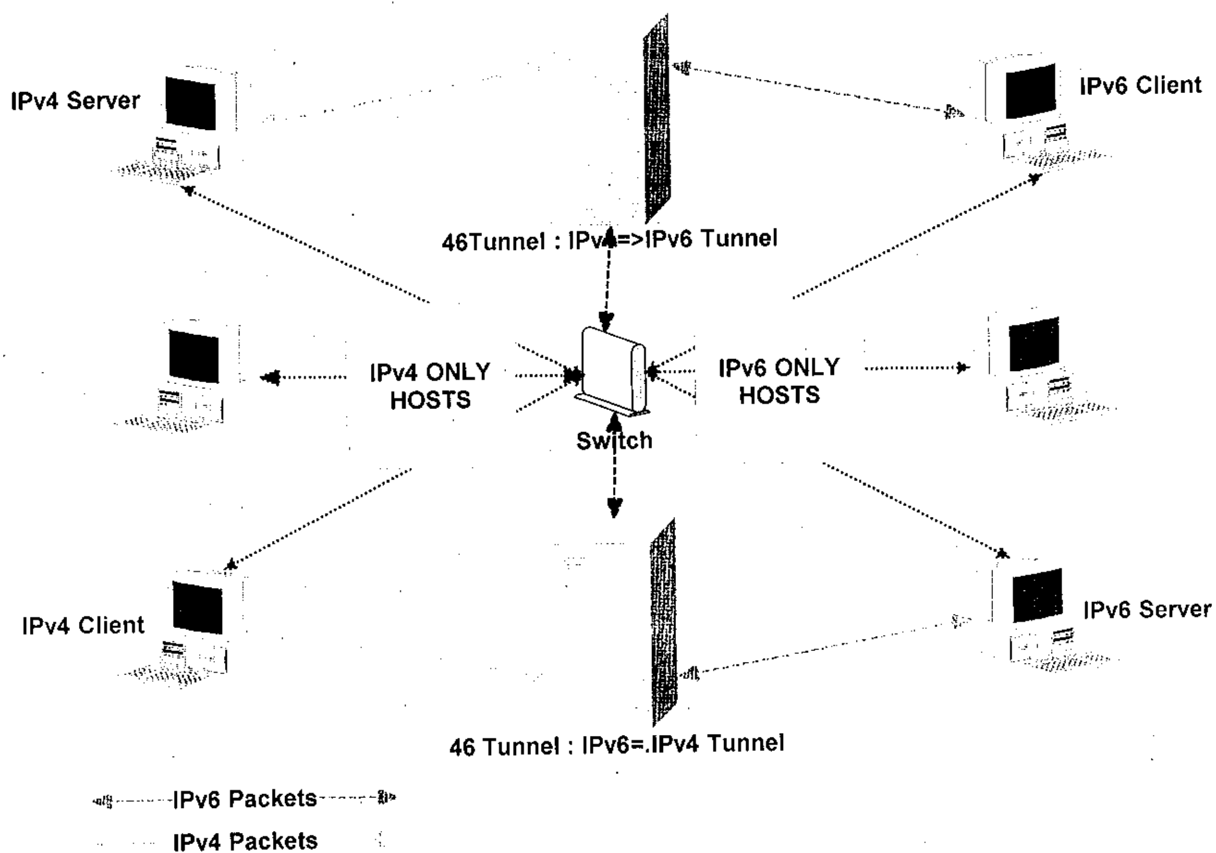


Figure 35 : Hardware Setup

- Minimally 3 nodes are connected via a network switch.
- 46tunnel software is installed on a node supporting both IPv4 and IPv6 protocols.

- 2nd node supports IPv4 only.
- 3rd node supports IPv6 only.
- 2nd and 3rd node are installed with .NET Framework version 2 or above to allow running of test application.
- All 3 nodes are running Microsoft Windows XP SP2.
- Data travels from node 2 to node 3 only through node 1 (with 46tunnel).
- Using the test application an IPv4 client is started on node 2.
- IPv6 server is started on node 3 using the test application.
- Client is fed the address (in IPv4 format) of the tunnel node 1.
- If client is IPv4 then tunnel type 4to6 is selected in the 46tunnel interface otherwise 6to4 tunnel is required.
- Path to a file is given through the client.
- The file is successfully received at the server.
- The test with an IPv4 server - IPv6 client is also successful.

5. DEPLOYMENT

Our "configured tunnel payload transfer" application – 46tunnel can be especially useful in the following cases:

- Alongside an IPv4 application so that it can connect to an IPv6 server.
- Alongside an IPv4 server so that it can accept queries from an IPv4 client.
- Alongside both the server and client to let IPv4 only and IPv6 clients and servers to interact.
- As a proxy server to relay packets of differing formats to different networks.

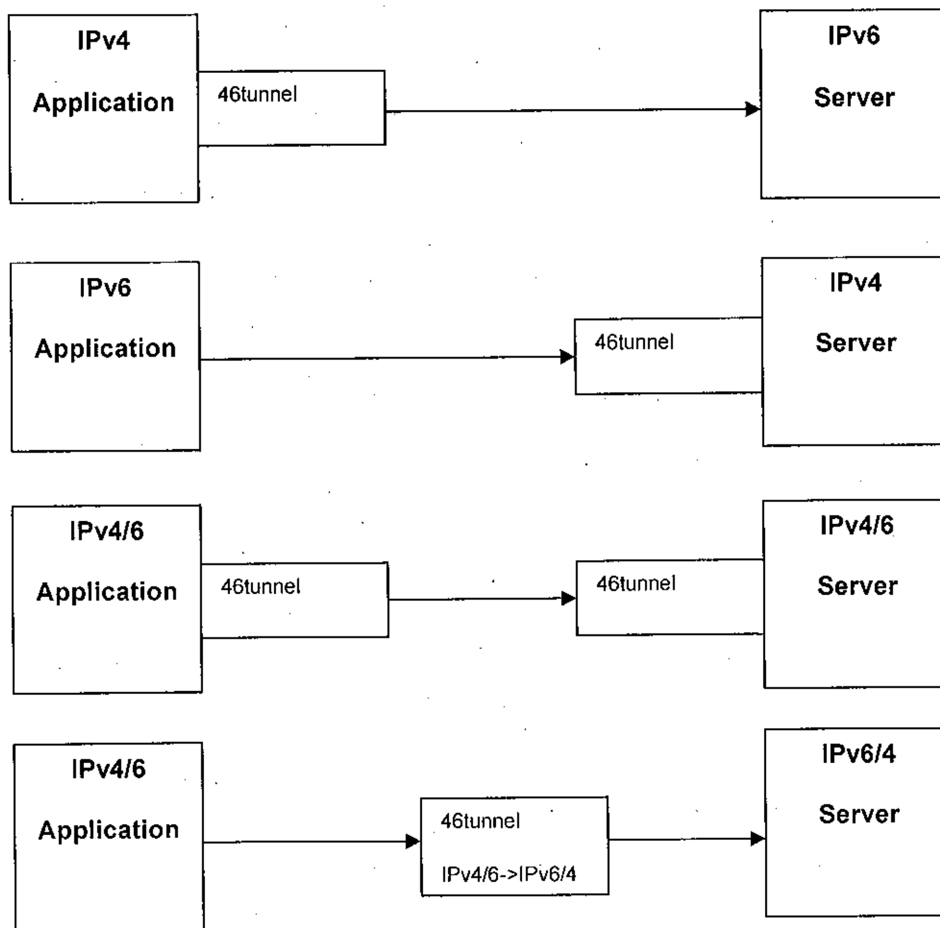


Figure 36: Deployment of the application in the real world

5.1 SCREEN SHOTS

5.1.1 Tunnel Application

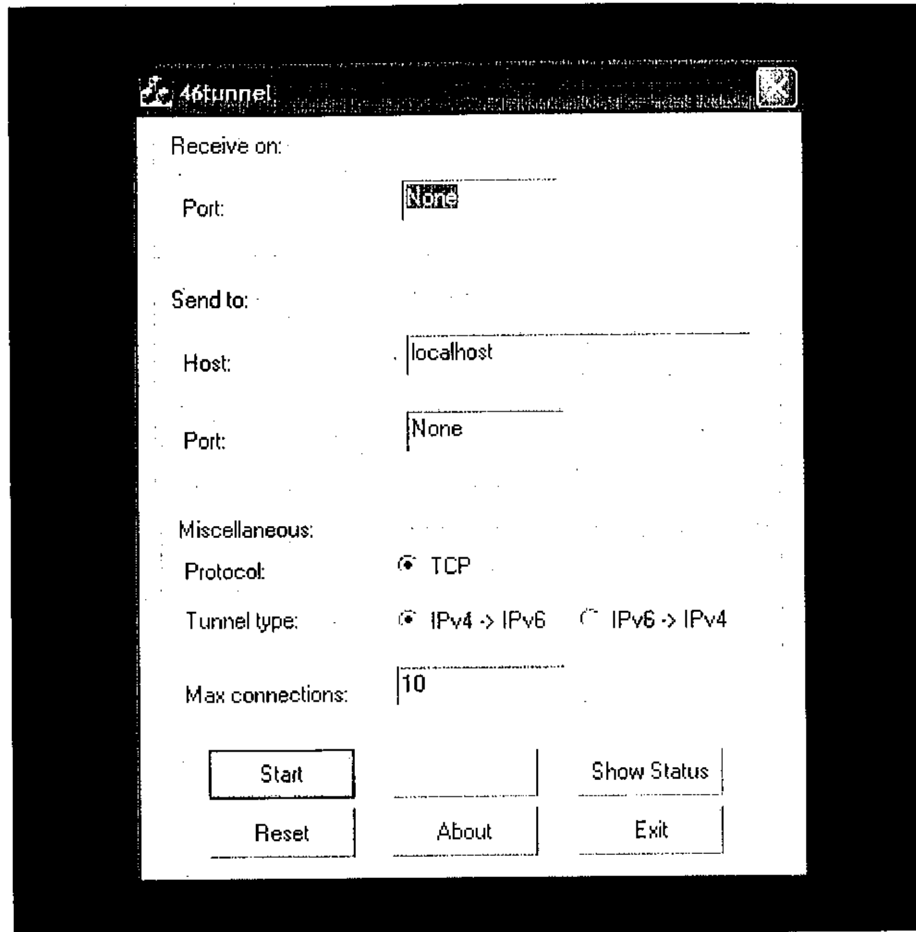


Figure 37:Initial Configuration of the Main Application

The fields in the application are:

Port: The Port on which the client will send data.

Host: The address of the server to which the data has to be transferred

Port: The port on which the server is listening.

Protocol: TCP is the only protocol supported.

Tunnel type: Whether the application will convert from IPv6 to IPv4 or vice versa.

Max Connections: The number of maximum connections it can receive on the listening port.

Start: Button to start the application with all the required fields filled in.

Stop: Button to stop the application from execution.

Show Status: To show the status of the application during execution.

Reset: To reset the fields of the application to the default values.

About: To see the version of the application.

Exit: To close the application.

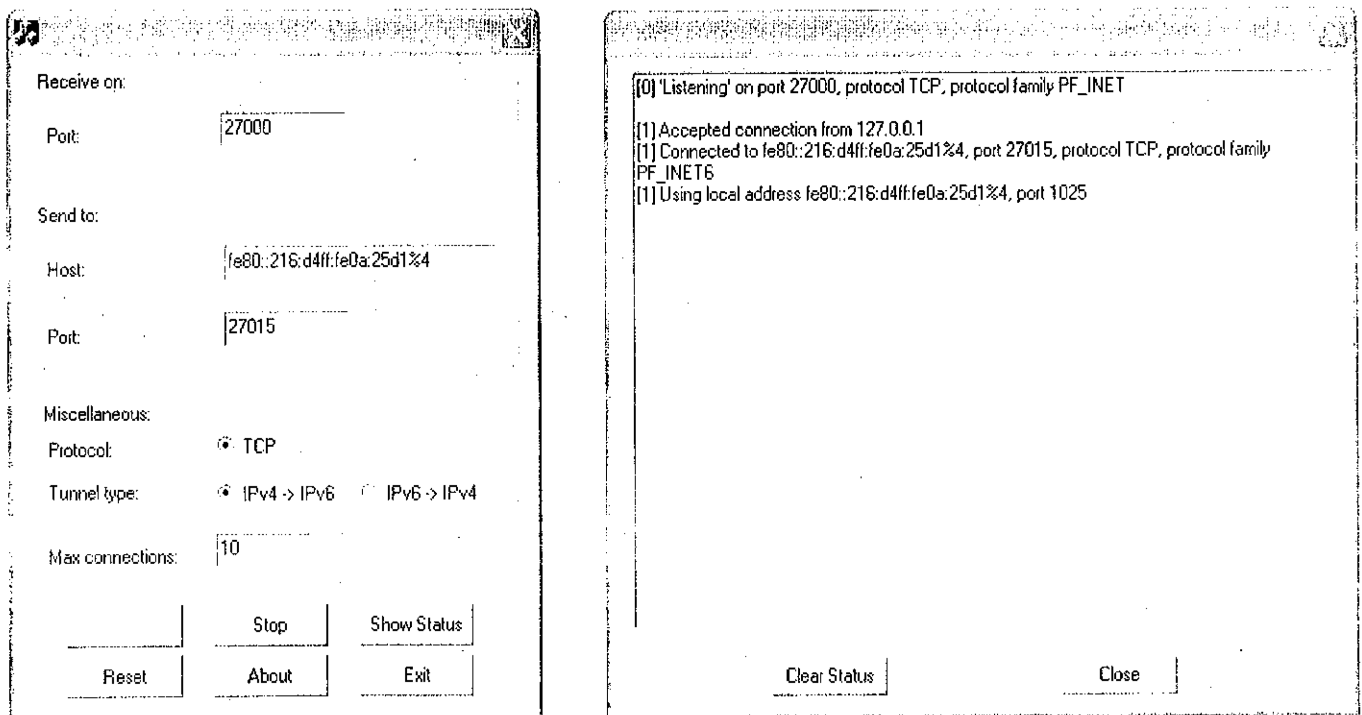


Figure 38 : After the start of the execution of the main application

The status window shows that the connection is accepted on port 27000 from the address 127.0.0.1

It is connected to server whose IPv6 address is also given with the required protocols.

It also shows the local IPv6 address and the local socket address

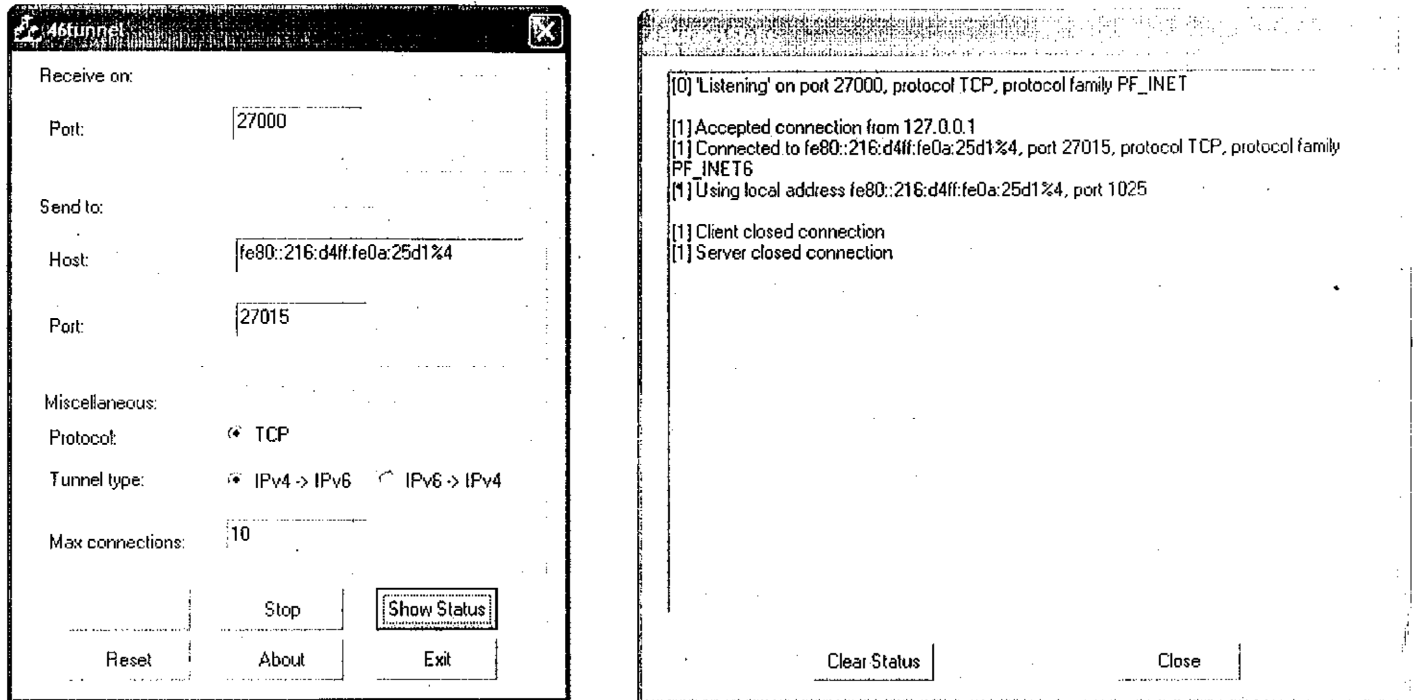


Figure 39 : The Status message after the completion of file transfer

The status message shows that the file transfer has taken place between the client and the server. And the connection is closed from both the ends.

5.1.2 Test Application

Figure 40 : Both ends of the test application

The Upper Window has an IPv6 Server. After the client starts, the server shows that it is connected to which client and at which port.The lower Window is an IPv4 Client which asks for a IPv4 server address to connect to.

Once the client gets connected it asks for the full path of the file which is to be sent.

5.2 Directories and file structure

5.2.1 Tunnel Application

46Tunnel1.0

46Tunnel.46b: Visual Studio 2005 self generated file

46Tunnel.aps: Visual Studio 2005 self generated file

46Tunnel.clw: Visual Studio 2005 self generated file

46Tunnel.cpp:Defines the class behavior and GUI for the application

46Tunnel.dsp: Visual Studio 2005 self generated file

46Tunnel.dsw: Visual Studio 2005 self generated file

46Tunnel.h:Main Header file for the application

46Tunnel.ncb: Visual Studio 2005 self generated file

46Tunnel.opt: Visual Studio 2005 self generated file

46tunnel.46b: Visual Studio 2005 self generated file

46tunnel.plg: Visual Studio 2005 self generated file

46tunnel.rc: Visual Studio 2005 self generated file

46tunnel.sln:Solution file of Visual Studio 2005

46tunnelDlg.cpp:Implementation of the main GUI window class

46tunnelDlg.h:Header file with class definition

NetworkInterface.cpp: Implementation of tunnel

NetworkInterface.h: Definition of class and structures used

resource.h: Visual Studio 2005 self generated file

Status.cpp: Implementation of the status window

Status.h: Definition of the class used

Debug: This directory contains all the files generated on build for debugging purpose.

Release:This directory contains all the files generated after debugging,it also contains the main **executable** file named **46tunnel.exe**

5.2.2 Test Application

FileTransfer

Program.cs: Source code for the test application

5.3 Environment for compilation and linking

5.3.1 Tunnel Application

Windows 98 or above, Visual Studio 2005,.Net Framework v 2.0 or above.

5.3.2 Test Application

Windows 98 or above, Visual Studio 2005,.Net Framework v 2.0 or above.

6. CONCLUSION

The Software is currently a prototype level "proof-of-concept" application with primitive optimizations and only basic functionality. The scope and scale of this application can be extended by future developers. The potential areas of extension are listed below.

1. The application can be modified to run as a gateway application, which parses each packet flowing in a network and the IPv4-IPv6 transition is seamless for the users of the network.

2. The application can be heavily optimized to handle thousands of concurrent queries. The delay in conversion of packet format can be speeded up by using low level libraries and raw-socket programming.

3. The application can be suitably changed to listen to some/all popular ports so that no user faces a transition incompatibility using his/her favorite application.

4. The application can also be implemented as a service in Microsoft Windows 2003 Server or as a daemon in UNIX/Linux flavor of operating systems so it becomes less obtrusive and requires little memory and CPU power.

5. The GUI can also be redone to increase usability and aesthetics of the application.

6. The application can also be archived inside an installer for easy installing and running of application without the hassles for various requirements.

7. BIBLIOGRAPHY

Books

- Mickey Williams and David Bennett, *Visual C++ 6 Unleashed.* Sams. 2000
- Richard C. Leinecker, *Visual C++ 6 Bible*, Hungry Minds. 1998
- Herbert Schildt, *A Complete Reference to Visual C++ 6*, Osborne/McGraw-Hill. 1998
- Jesse Liberty, *Programming C#: Building .NET Applications with C#*, O'Reilly. 2005

Research Papers

- R. Callon, D. Haskin, "*Routing Aspects of IPv6 Transition*", RFC 2185, September 1997.
- B. Carpenter, C. Jung, "*Transmission of IPv6 over IPv4 Domains without Explicit Tunnels*", RFC2529, March 1999.
- R. Rockell, R. Fink, "*6Bone Backbone Routing Guidelines*", RFC 2772, February 2000.
- Durand, P. Fasano, I. Guardini, D. Lento, "*IPv6 Tunnel Broker*", RFC3053, February 2001
- B. Carpenter, K Moore, "*Connection of IPv6 Domains via IPv4 Clouds*", RFC3056, February 2001
- J. Hagino, K. Yamamoto, "An IPv6-to-IPv4 transport relay translator", RFC3142, June 2001.