國立交通大學

資訊工程學系

碩士論文

IPv6 隧道之船蟲機制的效能分析

A Performance Study on Teredo IPv6 Tunneling Mechanism

研 究 生：黃祥鳴

指導教授：林一平　教授

　　　　　吳坤熹　教授

中 華 民 國 九 十 四 年 六 月

IPv6 隧道之船蟲機制的效能分析

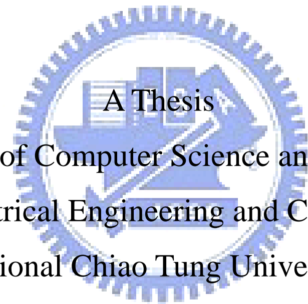
A Performance Study on Teredo IPv6 Tunneling Mechanism

研 究 生：黃祥鳴　　Student：Shiang-Ming Huang

指導教授：林一平　　Advisor：Yi-Bing Lin

吳坤熹　　Quincy Wu

國 立 交 通 大 學
資 訊 工 程 學 系
碩 士 論 文



A Thesis

Submitted to Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science and Information Engineering

June 2005

Hsinchu, Taiwan, Republic of China

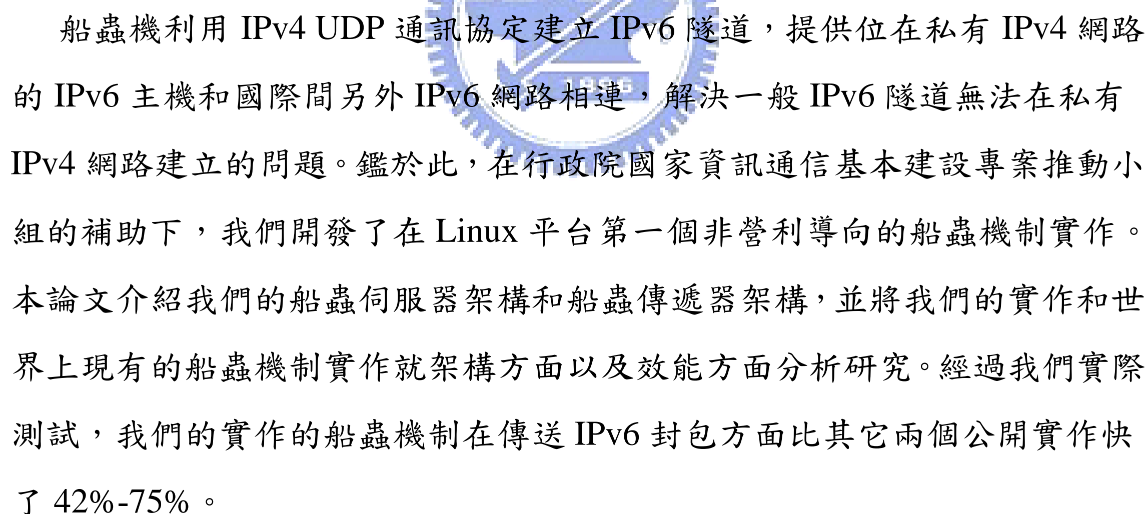
中華民國九十四年六月

IPv6 隧道之船蟲機制的效能分析

學生：黃祥鳴　　　　　　　　　　　指導教授：林一平 教授
吳坤熹 教授

國立交通大學資訊工程學系碩士班

摘　　　要



船蟲機利用 IPv4 UDP 通訊協定建立 IPv6 隧道，提供位在私有 IPv4 網路的 IPv6 主機和國際間另外 IPv6 網路相連，解決一般 IPv6 隧道無法在私有 IPv4 網路建立的問題。鑑於此，在行政院國家資訊通信基本建設專案推動小組的補助下，我們開發了在 Linux 平台第一個非營利導向的船蟲機制實作。本論文介紹我們的船蟲伺服器架構和船蟲傳遞器架構，並將我們的實作和世界上現有的船蟲機制實作就架構方面以及效能方面分析研究。經過我們實際測試，我們的實作的船蟲機制在傳送 IPv6 封包方面比其它兩個公開實作快了 42%-75%。

A Performance Study on Teredo IPv6 Tunneling Mechanism

Student：Shiang-Ming Huang

Advisors：Prof. Yi-Bing Lin
Prof. Quincy Wu

Department of Computer Science and Information Engineering
National Chiao Tung University

ABSTRACT

By tunneling IPv6 packets over IPv4 UDP, Teredo supports IPv6 hosts located within private IPv4 networks for connection with external IPv6 networks. Under National Information and Communication Initiative in Taiwan, we have developed the first non-commercial Linux-based Teredo mechanism. In this thesis, we describe our implementation of Teredo server and Teredo relay, and compare our solution with other Teredo implementations in the public domain. Our study indicates that our solution reduces the tunneling overhead and transmission delay over two other implementations by 42%-75%.

Acknowledgement

This thesis is the result of two years of work whereby I have been accompanied and supported by many people. It is a pleasant aspect that I have now the opportunity to express my gratitude for all of them.

I would first like to thank Prof. Yi-Bing Lin (林一平) and Prof. Quincy Wu (吳坤熹) for their guidance, insight and support throughout the preparation of this thesis.

I would like to thank Dr. Whai-En Chen (陳懷恩), Meng-Hsun Ts'ai (蔡孟勳), Wei-Che Huang (黃偉哲) and Jui-Hung Weng (翁瑞鴻) for the helpful discussions about the design of Teredo relay testing environment.

I would also like to thank my master committee members, Prof. Ai-Chun Pang (逄愛君) and Prof. Shun-Ren Yang (楊舜仁) for their helpful comments and suggestions.

My colleagues of Laboratory 117 all gave me the feeling of being at home at work. Many thanks for being your colleague.

I feel a deep sense of gratitude for my parents who formed part of my vision and taught me the good things that really matter in life. I am grateful for my three sisters for rendering me the sense and the value of brotherhood. I am glad to be one of them.

Finally, I would like to thank all my friends who inspire me or make me feel confidence to accomplish this thesis.

Shiang-Ming Huang
June 2005

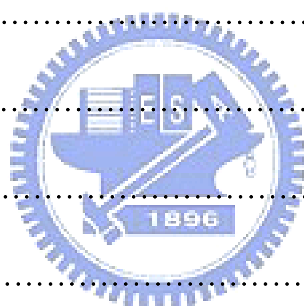
Contents

摘要……………………………………………………………………………….………...i

ABSTRACT……………………………………………………………….…………ii

Acknowledgement…………………………………………………………………iii

Contents…………………………………………………………………………....iv



List of Figures……………………………………………………………………...vi

List of Tables……………………………………………………………………...vii

Chapter 1 Introduction……………………………………………………………...1

Chapter 2 The Teredo Mechanism…………………………………………………3

2.1 Teredo Architecture…………………………………………….……3

2.2 Address Mapping on NAT……………………………………………...4

2.3 Teredo Addressing…………………………………………………………6

2.4 Packet Delivery through Teredo………………………………..…………8

Chapter 3 Three Teredo Implementations……………………………….……...10

3.1 Teredo Server……………………………………………………………11

3.1.1 NICI-Teredo Server…………………………………………11

3.1.2 6WIN-Teredo Server and Miredo-Teredo Server…………...12

3.2 Teredo Relay………………………………………………….……..13

3.1.1 NICI-Teredo Relay……………………………………………..13

3.2.2 6WIND-Teredo Relay…………………………………………15

3.2.3 Miredo-Teredo Relay…………………………………………..17

Chapter 4 Performance Evaluation of Teredo Relay Implementations..…………..20

4.1 Measurement Environment……………………………………………..20

4.2 Experimental Result…………………………………………….……...21

Chapter 5 Conclusions………………………………………………………………25

Bibliography…………………………………………………………………………26

Appendix A Decapsulation Latency Histograms……………………………………28

Appendix B The NICI-Teredo Server Program…………………………………...30

B.1 teredo_server.h…………………………………………………30

B.2 teredo_server.c…………………………………………………31

Appendix C The NICI-Teredo Relay Program……………………………….…...39

C.1 teredo.h……………………………………………………….….…39

C.2 bubble.h……………………………………………………………..41

C.3 teredo.c…………………………………………………………..….45

List of Figures

Figure 1. Teredo architecture……………………………………………………………4

Figure 2. An entry in the address mapping table…………………………………………5

Figure 3. Teredo IPv6 address format………………………………………………………6

Figure 4. Communication between a Teredo client and an IPv6 host……………………8

Figure 5. Software architecture of the NICI-Teredo server………………………………12

Figure 6. Software architecture of the NICI-Teredo relay…………………………………14

Figure 7. Software architecture of the 6WIND-Teredo relay………………………………16

Figure 8. Software architecture of the Miredo-Teredo relay………………………………18

Figure 9. The test environment………………………………………………………………21

Figure 10. Encapsulation latency histograms of the Teredo relays (1280 bytes)………22

Figure 11. Encapsulation latency histograms of the Teredo relays (512 bytes)…………23

Figure 12. Encapsulation latency histograms of the Teredo relays (64 bytes)…………..23

Figure 13. Decapsulation latency histograms of the Teredo relays (1280 bytes)………28

Figure 14. Decapsulation latency histograms of the Teredo relays (512 bytes)…………29

Figure 15. Decapsulation latency histograms of the Teredo relays (64 bytes)…………..29

List of Tables

Table 1. Average processing latency..…………………………….…….….……...22

Chapter 1
Introduction

After three decades of evolution in Internet technologies, IETF (Internet Engineering Task Force) has developed Internet Protocol version 6 (IPv6) as the next generation Internet protocol. In comparison with the existing Internet Protocol version 4 (IPv4), IPv6 provides larger address space, more efficient routing mechanism, better support for security and quality of service (QoS).

During IPv6 deployment, many existing IP networks remain to support IPv4 only. Because each site may deploy IPv6 in incremental fashion, it is unrealistic to assume a single flag day to upgrade all IPv4 networks to IPv6. To facilitate the transition for IPv4 to IPv6 migration, *tunneling* techniques are utilized to carry IPv6 traffic through IPv4 networks [1].

In existing IPv6-in-IPv4 tunneling mechanisms such as configured tunnel & automatic tunnel [1], 6to4 tunnel [2] and tunnel broker [3], both end points of a tunnel must possess public IPv4 addresses. Although public IPv4 addresses may be available in some typical scenarios, many Internet service providers, especially WLAN (wireless local area network) and GPRS (general packet radio service) [4], may only provide private IPv4 addresses to their customers, and NAT (network address translation) [5] is required to establish Internet connectivity. Hence,

IPv6 users within private IPv4 networks would not be able to establish tunnels to public IPv6 networks. This issue turns out to be one of the major obstacles in IPv6 deployment.

Several IPv6 tunneling solutions for private IPv4 networks with NAT have been proposed, including virtual private network (VPN) and user datagram protocol (UDP) tunnel [6]. These solutions provide IPv6 connectivity for private hosts, but require manual configuration at the user end of a tunnel. This configuration task is not transparent to users, and is not easy for novice users. Therefore, it is not suitable for large private IPv4 networks. Moreover, in these approaches, only one static tunnel server is assigned to relay all IPv6 packets of a host in the private network. This tunnel server may potentially become the bottleneck, and it is very likely that the traffic follows a "dog leg" route from the source to the tunnel server and then to the destination, resulting in a non-optimal routing path. The above issues can be resolved by Teredo [7], an automatic tunneling mechanism with the capability to traverse NATs[1].

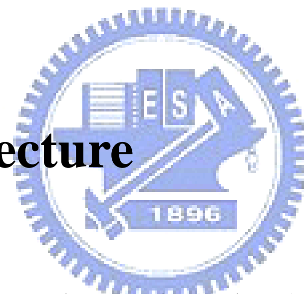
Under the support of NICI (National Information and Communication Initiative), we have developed the first non-commercial Linux-based Teredo to speed up the IPv6 deployment. This thesis describes our Teredo implementation and then compares it with other public-domain Teredo solutions.

---

[1] Recently, an enhanced model of UDP tunnel called "Silkroad" [8] was proposed to alleviate the bottleneck issue. However, some critical algorithms of Silkroad are still missing at the time when this thesis is written.

Chapter 2
The Teredo Mechanism

This chapter first introduces Teredo architecture and address mapping on NAT, and then describes how Teredo applies this address mapping technique to help tunneling IPv6 through NAT. Finally; we conclude this chapter with an example that shows how an IPv6 packet is delivered through Teredo.


ES 1896

2.1 Teredo Architecture

By tunneling IPv6 packets over IPv4 UDP through NATs, Teredo provides IPv6 connectivity for IPv6 hosts within private IPv4 networks. The Teredo architecture is illustrated in Figure 1, which consists of a Teredo server (Figure 1 (a)), several Teredo clients (Figure 1 (b)) and Teredo relays (Figure 1 (c)). A Teredo client is implemented at an IPv6 host in a private IPv4 network (Figure 1 (d)) that connects to public IPv4 network (Figure 1 (e)) through NAT (Figure 1 (f)). A Teredo server assists a Teredo client to obtain an IPv6 address for IPv6 network access. For IPv6 packets sent from the IPv6 network, the Teredo relay encapsulates these packet in IPv4 UDP and forwards them to the destination Teredo client in a private IPv4 network. In the reverse direction, the Teredo relay decapsulates IPv4 UDP packets sent from the Teredo client to the IPv6 network (Figure 1 (g)). To broadcast its identity, every Teredo

relay advertises an IPv6 address prefix 3FFE:831F::/32 to the IPv6 network (path ① in Figure 1). Through the advertisement, the IPv6 hosts (Figure 1 (h)) select appropriate Teredo relays such that all IPv6 packets sent from these IPv6 hosts to a Teredo client are routed to Teredo relays closest to the packet sources. Therefore the traffic load to a Teredo client can be dynamically adjusted among Teredo relays with optimal routing.

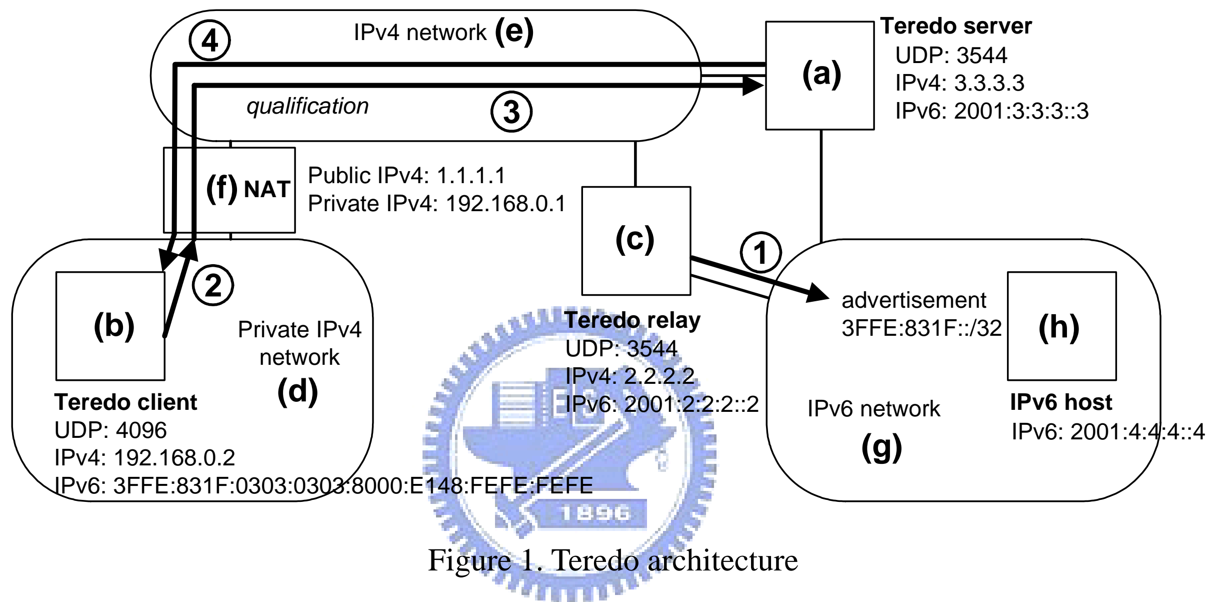


Figure 1. Teredo architecture

2.2 Address Mapping on NAT

In Figure 1, the IPv4 address of the Teredo server is 3.3.3.3. The NAT is equipped with two network interfaces: the WAN interface (to the public network) has the public IPv4 address 1.1.1.1, and the LAN interface (to the private network) has the private IPv4 address 192.168.0.1. The Teredo client is assigned the private IPv4 address 192.168.0.2. The NAT performs private-public address translation for all pass-through packets according to an

address mapping table. Suppose that the NAT is a full cone NAT[2] [9]. The fields of an entry in the NAT address mapping table are illustrated in Figure 2. In this table, the "private IPv4 address" and the "private port" fields store the private transport address of the private end (i.e. IPv4 address plus TCP/UDP port, whose values are 192.168.0.2 and 4096 for the example in Figure 1). The "transport protocol" field stores the transport protocol type (TCP or UDP). The "public IPv4 address" and the "public port" fields store the public transport address assigned by the NAT. The values in these fields (IPv4 address 1.1.1.1 and port 7863 in this example) are used to replace the private transport address. This public transport address for a private IPv4 host is crucial for IPv4 UDP packets passing through the NAT. When a host in the public IPv4 network sends an IPv4 UDP packet to this transport address (1.1.1.1:7863), the NAT dispatches this packet to the designated private IPv4 host with the help of the address mapping table.

transport protocol	private IPv4 address	private port	public IPv4 address	public port

Figure 2. An entry in the address mapping table

[2] When an internal host within a private network sends a packet to an external host, a NAT maps the private transport address of the internal host to a unique public transport address. Thereafter, a public host can send packets to the private host by delivering them to the mapped public transport address. However, different types of NATs have different rules to handle incoming packets. The full cone NAT allows packets from all public hosts to pass through, while a restricted cone NAT allows an external host (with IP address X) sending a packet to the private transport address only if the private transport address had previously sent a packet to IP address X.

2.3 Teredo Addressing

As an automatic tunneling mechanism, Teredo embeds the NAT traversal information in its 128-bit IPv6 address. A Teredo IPv6 address format consists of the following fields (see Figure 3).

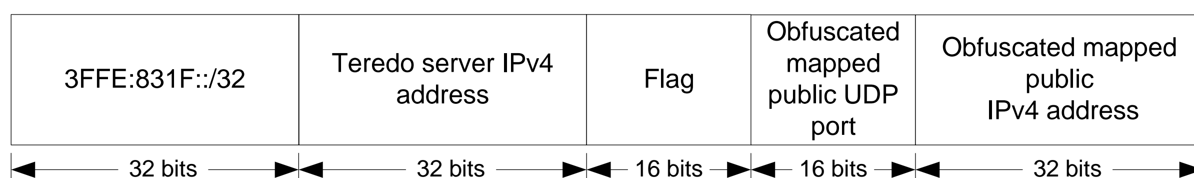


Figure 3. Teredo IPv6 address format

The “3FFE:831F::/32” field specifies the IPv6 address prefix for Teredo. The “Teredo server IPv4 address” field indicates the IPv4 address of a Teredo server (e.g. 3.3.3.3 in Figure 1). The “Flag” field indicates the type of NAT (full cone or not) [9]. The “Obfuscated mapped public UDP port” and the “Obfuscated mapped public IPv4 address” fields indicate the public trasnport address assigned by the NAT. This transport address is mapped to the Teredo client’s private transport address. The obfuscation mechanism is needed because some NAT products provide “generic” application layer gateway (ALG) functionality. The generic ALG hunts for IPv4 addresses, either in text or binary formats within a packet, and rewrites them if they match a binding in the address mapping table. When this “smart NAT” handles the payload of pass-through IPv4 packets, it translates any occurrence of IPv4 address in the payload that matches the address to be translated in the IPv4 header (or translates any occurrence of port number in the payload that matches the port to be translated in the TCP/UDP header). Such action certainly interferes with normal Teredo operations. To prevent the smart NAT from

modifying the Teredo IPv6 addresses in the encapsulated IPv4 UDP packets, obfuscation performs bitwise XOR operation on the original value with 1 to protect it.

At start-up, a Teredo client obtains a Teredo IPv6 address from the Teredo server by performing the *qualification* procedure (path ②→③ in Figure 1). This procedure detects the NAT type and informs the Teredo client of the mapped public transport address. As long as the Teredo clients obtain Teredo IPv6 addresses, they are able to communicate with the IPv6 network with the help of Teredo servers and Teredo relays. In Figure 1, the Teredo client uses UDP port 4096 to request a Teredo IPv6 address from the Teredo server. When this packet arrives at the NAT (path ② in Figure 1), the NAT dynamically allocates an available UDP port (e.g. 7863 in this example) for this connection, and creates an entry in the address mapping table with protocol type UDP, private transport address 192.168.0.2:4096 and public transport address 1.1.1.1:7863. Then the UDP packet is sent to the Teredo server (path ③ in Figure 1). The Teredo server calculates the Teredo IPv6 address for this Teredo client by determining the value of each field as follows:

- Prefix (32 bits) = 0x3FFE831F
- Teredo server IPv4 address (32 bits) = 3.3.3.3 = 0x03030303
- Flag (16 bits) = 0x8000 (full cone NAT)
- Obfuscated mapped public UDP port (16 bits) = 7863 $\oplus$ 0xFFFF = 0x1EB7 $\oplus$ 0xFFFF = 0xE148
- Obfuscated mapped public IPv4 address (32 bits) = 1.1.1.1 $\oplus$ 0xFFFFFFFF = 0x01010101 $\oplus$ 0xFFFFFFFF = 0xFEFEFEFE

Therefore, the Teredo IPv6 address assigned to this Teredo client by the Teredo server (path ④ in Figure 1) is 3FFE:831F:0303:0303:8000:E148:FEFE:FEFE.

2.4 Packet Delivery through Teredo

After qualification, communication between the Teredo client and an IPv6 host can be established.

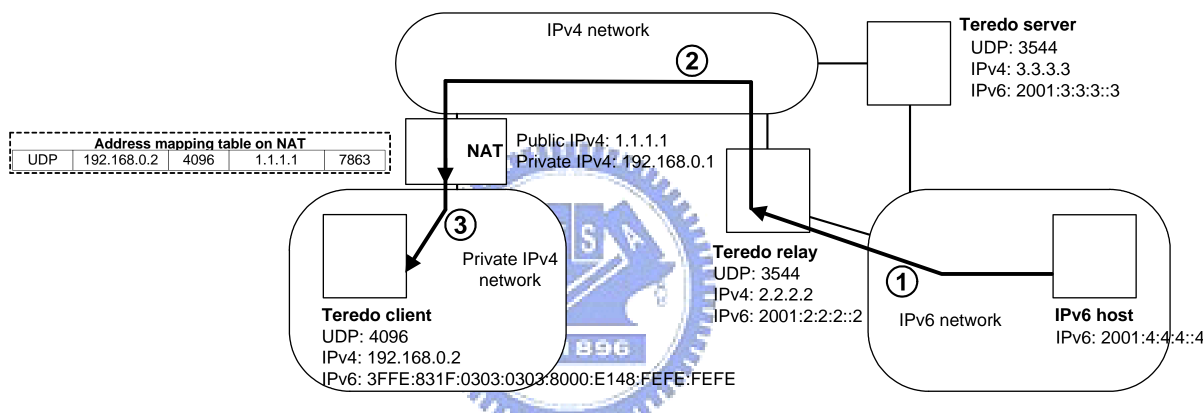


Figure 4. Communication between a Teredo client and an IPv6 host

Figure 4 illustrates how an IPv6 packet is delivered from an IPv6 host to a Teredo client with the following steps (see path ①→②→③ in Figure 4).

Step 1. The IPv6 packet is sent from the IPv6 host to the Teredo relay. (The IPv6 host selects this Teredo relay according to the advertisement; see path ① in Figure 1.)

Step 2. The Teredo relay encapsulates the IPv6 packet in an IPv4 UDP packet. The source address is the IPv4 address of the Teredo relay (2.2.2.2), and the source UDP port is 3544. The destination IPv4 address and UDP port of this packet are determined based on the IPv6 address of this Teredo client as follows. The destination address is the restored value derived from the 32-bit "Obfuscated mapped public IPv4 address" field of destination IPv6 address (0xFEFEFEFE$\oplus$0xFFFFFFFF = 0x01010101 = 1.1.1.1), and the destination UDP port is the restored value derived from the 16-bit "Obfuscated mapped public UDP port" field of destination IPv6 address (0xE148$\oplus$0xFFFF = 0x1EB7 = 7863). The Teredo relay sends the IPv4 UDP packet to the NAT (1.1.1.1:7863) through the IPv4 network.

Step 3. When the NAT receives this IPv4 UDP packet, it translates the destination IPv4 address and UDP port from 1.1.1.1:7863 to 192.168.0.2:4096 according to the address mapping table, and then sends it to the Teredo client in the private IPv4 network.

Upon receipt of the packet, the Teredo client decapsulates the IPv4 UDP packet to obtain the IPv6 packet. With the above steps, IPv6 packets sent from the IPv6 host can successfully pass through the NAT. This example assumes a full cone NAT server. Details of packet transmission for other types of NATs can be found in the Teredo draft [7].

Chapter 3
Three Teredo Implementations

We implemented NICI-Teredo [10] on Linux in year 2003. NICI-Teredo supports the Teredo server and Teredo relay functions that can be installed on a single host or independently on multiple hosts. The Teredo server function is implemented as a user-level daemon, which is illustrated in Figure 5 (a) and the source code is listed in Appendix B. The Teredo relay function is developed with a combination of a user-level program and a kernel-level module as illustrated in Figure 6 (a). The user-level programs deal with the NICI-Teredo configuration, while the kernel-level module supports high-speed IPv6 packet relaying (the source code is listed in Appendix C).

In the same year that we implemented NICI-Teredo, an independent Teredo implementation for FreeBSD was also developed by 6WIND, LIP6 and Euronetlab [11]. In year 2004, Miredo-Teredo was developed on Solaris, FreeBSD, and Linux [12]. In this chapter, we illustrate the software architectures of these Teredo implementations with a focus on Teredo server and Teredo relay.

3.1 Teredo Server

3.1.1 NICI-Teredo Server

The NICI-Teredo server (Figure 5 (a)) invokes an IPv6 raw Ethernet socket (Figure 5 (b)) and two IPv4 UDP sockets (Figure 5 (c)) at the Linux kernel to send and receive packets. The NICI-Teredo server consists of four components. The **packet processor** (Figure 5 ①) handles IPv6 packet encapsulation and IPv4 UDP packet decapsulation. The **dispatcher** (Figure 5 ②) checks the IPv6 packets delivered from the **packet processor** and dispatches them to the proper functions. The **qualification function** (Figure 5 ③) performs the qualification procedure for Teredo clients. This function helps the Teredo client to discover the type of NAT and the mappped public transport address, as described in Chapter 2. The Teredo server interacts with the Teredo clients for qualification through path Ⓐ→Ⓑ→Ⓒ→Ⓓ→Ⓔ in Figure 5. For a Teredo client which attempts to communicate with an IPv6 host, the **ICMPv6 relay function** (Figure 5 ④) helps to locate a Teredo relay closest to the destination IPv6 host. This discovery task is achieved by the ICMPv6 echo request sent from the Teredo client to the destination IPv6 host through path Ⓐ→Ⓑ→Ⓕ→Ⓖ in Figure 5 and the response sent from the IPv6 host to the Teredo client following path ①→②→③ in Figure 4. From the response message, the Teredo relay is located for the communication between the Teredo client and the IPv6 host.

3.1.2 6WIND-Teredo Server and Miredo-Teredo Server

The 6WIND-Teredo server and Miredo-Teredo server have similar architecture design, and the details are omitted.

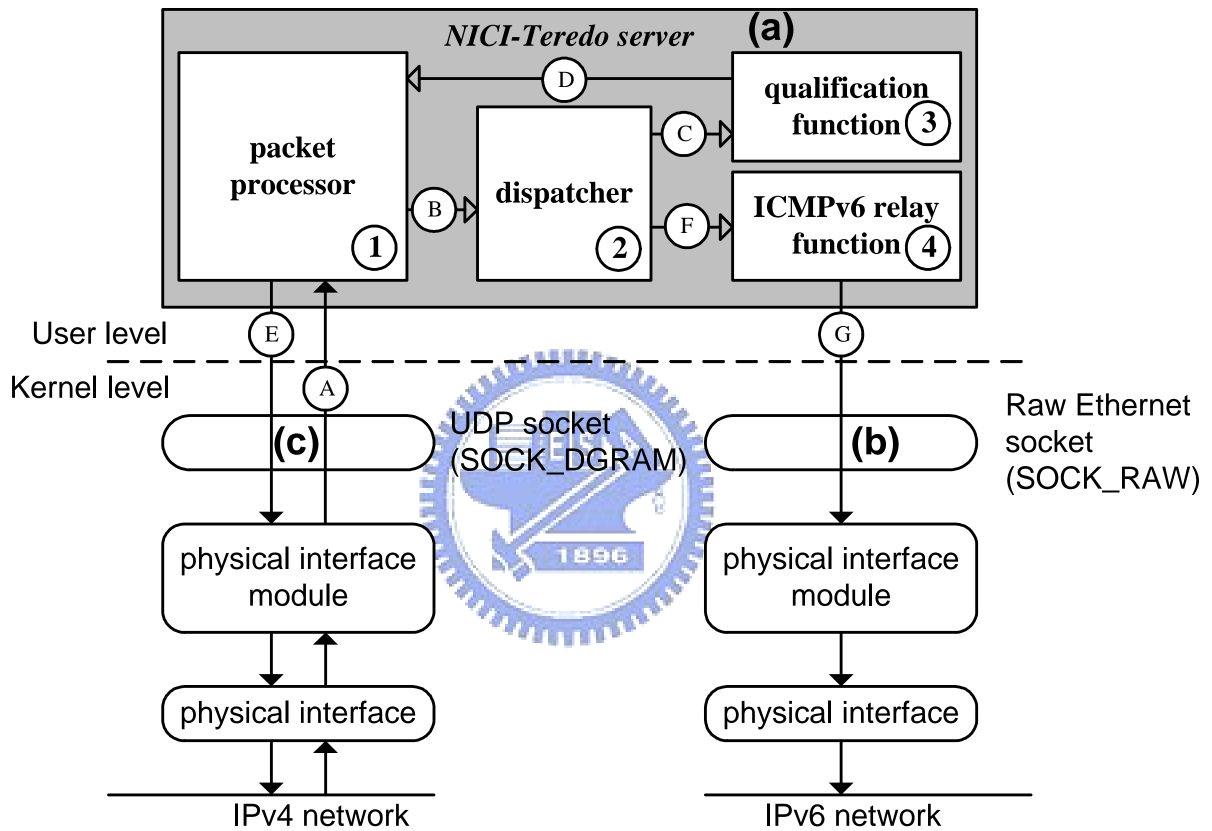


Figure 5. Software architecture of the NICI-Teredo server

3.2 Teredo Relay

3.2.1 NICI-Teredo Relay

The NICI-Teredo relay (Figure 6 (a)) provides the IPv6 packet relay function by utilizing the IPv6 and IPv4 forwarding mechanisms (Figures 6 (b) and (c)) at the Linux kernel. This Teredo relay consists of three modules. The **relay module** (Figure 6 ①) provides packet encapsulation and decapsulation functions by two callback functions, `udpip6_tunnel_xmit()` (Figure 6 Ⓐ) and `udpip6_rcv()` (Figure 6 Ⓑ), invoked by the IPv6 and the IPv4 forwarding mechanisms. From the IPv6 forwarding mechanism, the **relay module** receives IPv6 packets for IPv4 UDP encapsulation, and then passes them to the IPv4 forwarding mechanism by `IPTUNNEL_XMIT()`[3] (Figure 6 Ⓒ). The encapsulated packets are then delivered to the Teredo clients. In the reverse direction, the tunneled IPv4 UDP packets are also decapsulated into IPv6 by this module, and then passed to the IPv6 forwarding mechanism (i.e. via Linux function `netif_rx()`) for delivery to the IPv6 networks (Figure 6 Ⓓ). The **routing management module** (Figure 6 ②) maintains the packet forwarding plans at the Linux kernel. It configures the IPv6 forwarding plan to route the IPv6 packets to the Teredo clients (i.e. for the packets with the destination IPv6 addresses matching the IPv6 address prefix 3FFE:831F::/32) through the **relay module**, and configures the IPv4 forwarding plan to dispatch the IPv4 UDP packets from the Teredo clients to the

---

[3] According to the Linux programming convention, the uppercase name `IPTUNNEL_XMIT()` refers to a macro, while the lowercase name such as `udpip6_rcv()` refers to a function. The same convention is used in the 6WIND implementation on FreeBSD.

relay module. The **prefix advertisement module** Figure 6 ③) advertises the IPv6 address prefix 3FFE:831F::/32 to the IPv6 network so that the IPv6 packets destined to the Teredo clients can be routed to the nearest Teredo relay.

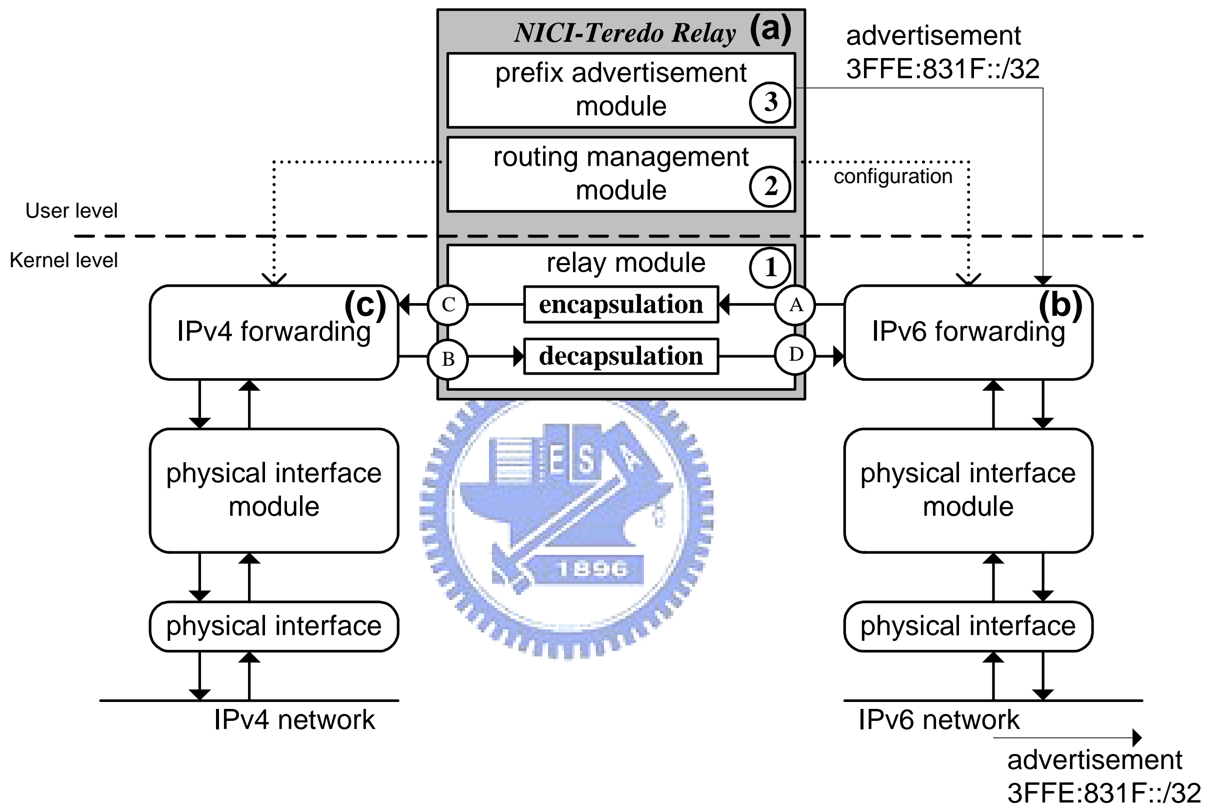


Figure 6. Software architecture of the NICI-Teredo relay

3.2.2 6WIND-Teredo Relay

6WIND-Teredo relay (Figure 7 (a)) provides IPv6 packet relay function by **netgraph**, a FreeBSD in-kernel networking subsystem. This Teredo relay relies on IPv6 forwarding mechanism (Figure 7 (b)) and socket functions (Figure 7 (c)) at the FreeBSD kernel to provide packet relay function. Unlike NICI-Teredo relay that utilizes a single kernel-level module to provide all packet processing functions, 6WIND-Teredo relay relies on the cooperation of several **netgraph** submodules and a kernel IPv4 UDP socket to handle the packets. 6WIND-Teredo relay consists of three components. The **netgraph module** (Figure 7 ①) utilizes three submodules to deals with IPv6 packet processing. The `netisr_dispatch()` and `ng_iface_output()` functions of the **ngN** submodule send and receive IPv6 packets; the `so_pru_sosend()` and `so_pru_soreceive()` functions of the **ksocket** submodule utilize a kernel IPv4 UDP socket to send and receive the encapsulated packets. These two **netgraph** submodules interact with each other through the **ng_teredo** submodule. Specifically, the **ng_teredo** submodule uses `NG_SEND_DATA()` and `ng_teredo_rcvdata()` to send and receive IPv6 packets between the **ngN** and the **ksocket** submodules. By the cooperation of these three submodules in the **netgraph module**, IPv6 packets received by the **ngN** submodule are written to the kernel's IPv4 UDP socket by the **ksocket** submodule, and vice versa. The IPv4 UDP socket provides packet encapsulation and decapsulation functions by delivering an IPv6 packet as the datagram through the socket. The **routing management module** (Figure 7 ②) configures the IPv6 forwarding plan and the interconnection between the **netgraph** submodules. The **prefix advertisement module** (Figure 7 ③) provides similar functions as the prefix advertisement module in the

NICI-Teredo relay, and the details are omitted.

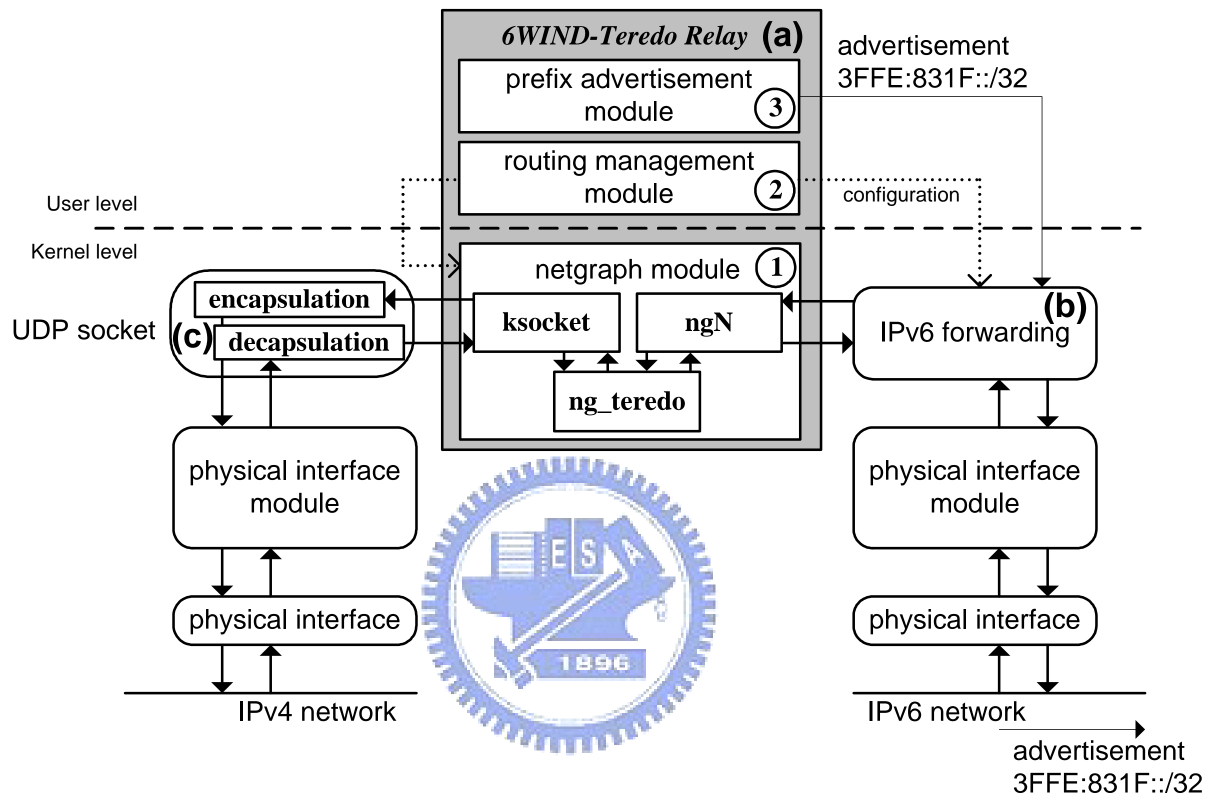


Figure 7. Software architecture of the 6WIND-Teredo relay

3.2.3 Miredo-Teredo Relay

The software architecture of the Miredo-Teredo relay (Figure 8 (a)) is quite different from the previous two implementations. Miredo-Teredo relay relies on IPv6 forwarding mechanism (Figure 8 (b)) and an IPv4 UDP socket (Figure 8 (c)) for packet processing. Unlike NICI-Teredo relay that uses a kernel-level packet relay module, Miredo-Teredo relay uses a user-level packet processing module to send and receive the encapsulated packets. This module delivers an IPv6 packet as the encapsulated datagram through the IPv4 UDP socket. Miredo-Teredo relay consists of four components. The **tun module** (Figure 8 ①) is built on the IPv6 protocol stack to send and receive IPv6 packets. The **packet processing module** (Figure 8 ②) creates and maintains an IPv4 UDP socket (Figure 8 (c)) for packet encapsulation and decapsulation. This module provides IPv6 packet relay function by using `memcpy()` and socket functions (`sendto()` and `recvfrom()`) for IPv6 packets delivery between the **tun module** and the IPv4 UDP socket (see Figures 8 Ⓐ and Ⓑ). This architecture is clearly different from those of NICI-Teredo relay and 6WIND-Teredo relay. On the other hand, the Miredo-Teredo relay **routing management module** (Figure 8 ③) and the **prefix advertisement module** (Figure 8 ④) are similar to those of NICI-Teredo relay and 6WIND-Teredo relay.

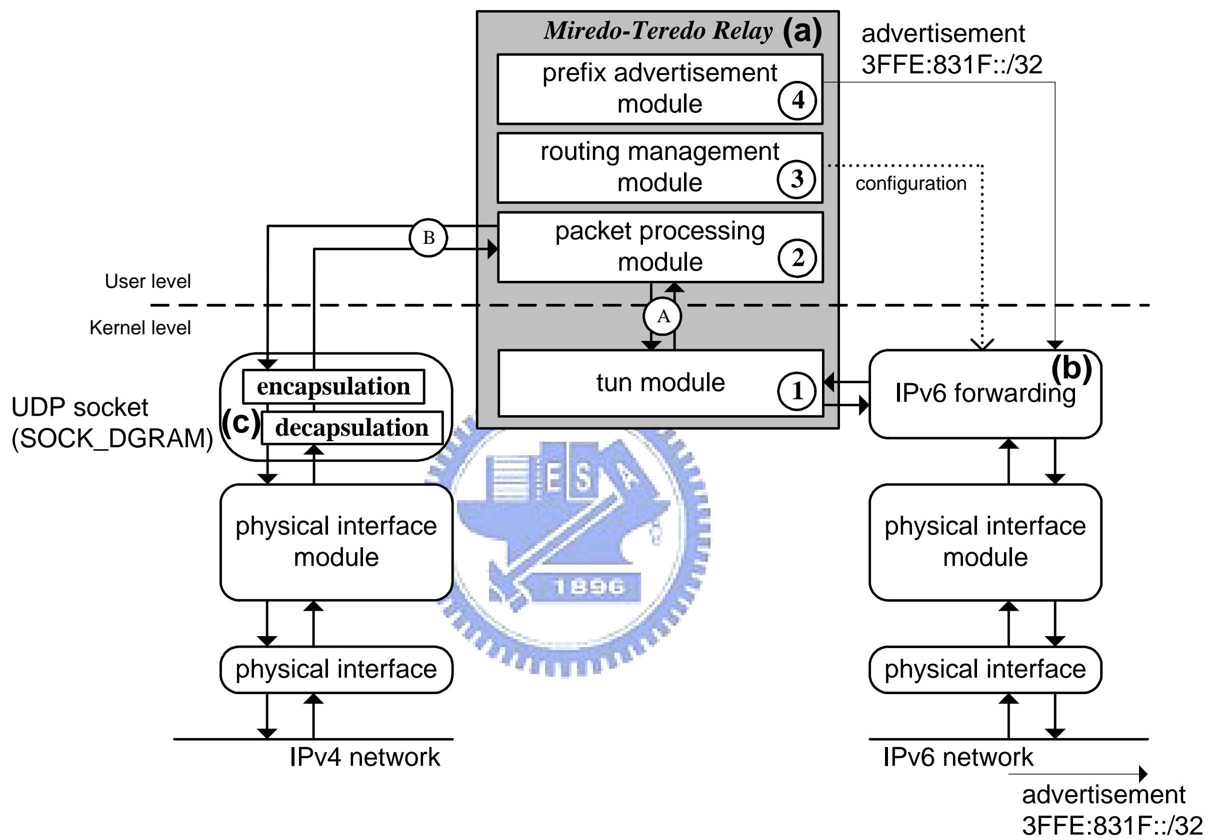


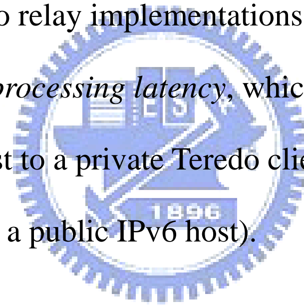
Figure 8. Software architecture of the Miredo-Teredo relay

The Teredo relay design significantly affects the packet transmission performance. The packet processing latency of NICI-Teredo relay is shorter because there are no packet copying operations between the kernel and the user levels. The performance comparison of these Teredo implementations will be elaborated in the next chapter.

As a final remark, the NICI-Teredo relay is implemented as a loadable kernel module [13]. Although kernel hacking effort is required in developing the NICI-Teredo relay, the installation process is very simple and the users do not need to modify or re-compile the Linux kernel.

Chapter 4
Performance Evaluation of Teredo Relay Implementations

Teredo relay handles large volume of network traffic and therefore is likely to be the bottleneck component in the Teredo mechanism. In this chapter, we investigate the performance of the three Teredo relay implementations described in the previous chapter. The output measurement is *packet processing latency*, which includes both packet encapsulation latency (from a public IPv6 host to a private Teredo client) and packet decapsulation latency (from a private Teredo client to a public IPv6 host).



4.1 Measurement Environment

The measurement environment consists of three hosts as illustrated in Figure 9. This environment follows the testing architecture in RFC 2544 [14] where a tester (Figure 9 (a)) is configured as both the Teredo client function and the IPv6 host. This IPv6 host runs on Redhat Linux 9 with an IPv4 UDP daemon to simulate the Teredo client function. The NAT (Figure 9 (b)) runs on Redhat Linux 9 with address mapping rules set by *iptables* [15]. The devices under test (DUTs) are the three Teredo relay implementations (Figure 9 (c)):

NICI-Teredo (version 0.3), 6WIND-Teredo (version 1.13) and Miredo-Teredo (version 0.3.0). Both NICI-Teredo relay and Miredo-Teredo relay run on Redhat Linux 9, while 6WIND-Teredo relay runs on FreeBSD 4.9. The hardware for the Teredo relay in Figure 9 is a personal computer with 1800+ AMD Athlon CPU, 256 MB SDRAM and two RealTek 8139 100BaseTx Ethernet cards.

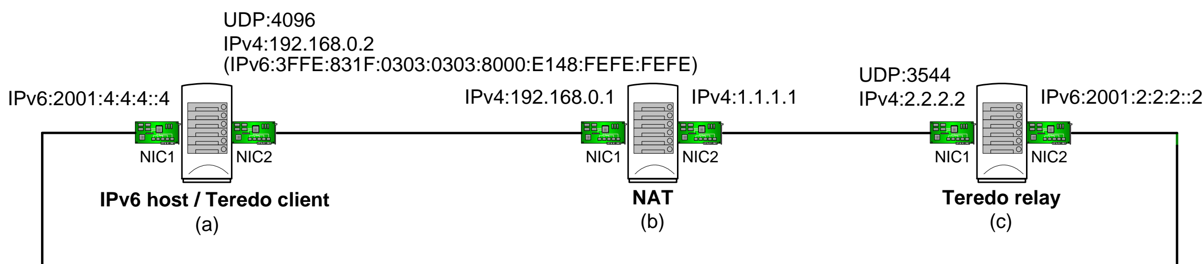


Figure 9. The test environment

In our measurement, a C program invoking *pcap* library [16] is used for catching the packet receiving and sending timestamps. We send one packet per second to the Teredo relay for encapsulation or decapsulation, and measure the packet processing latency. Tests are conducted with three different packet sizes (64, 512, and 1280 bytes, where the 1280-byte packet is the recommended IPv6 MTU size for Teredo [7]). Each test generates 10,000 packets to measure the encapsulation and decapsulation latencies.

4.2 Experimental Result

Figure 10 shows the 1280-byte packet encapsulation latency histograms of the three Teredo relays. The latency of the NICI-Teredo relay is clustered around 7-9µs and 12-13µs. The

latency of the 6WIND-Teredo relay is around 12-16μs. The latency of the Miredo-Teredo relay is clustered around 23-26μs and 33-39μs. The 1280-byte packet decapsulation latency histograms are similar to those in Figure 10, and the average packet processing latency of the three Teredo relays are listed in Table 1.

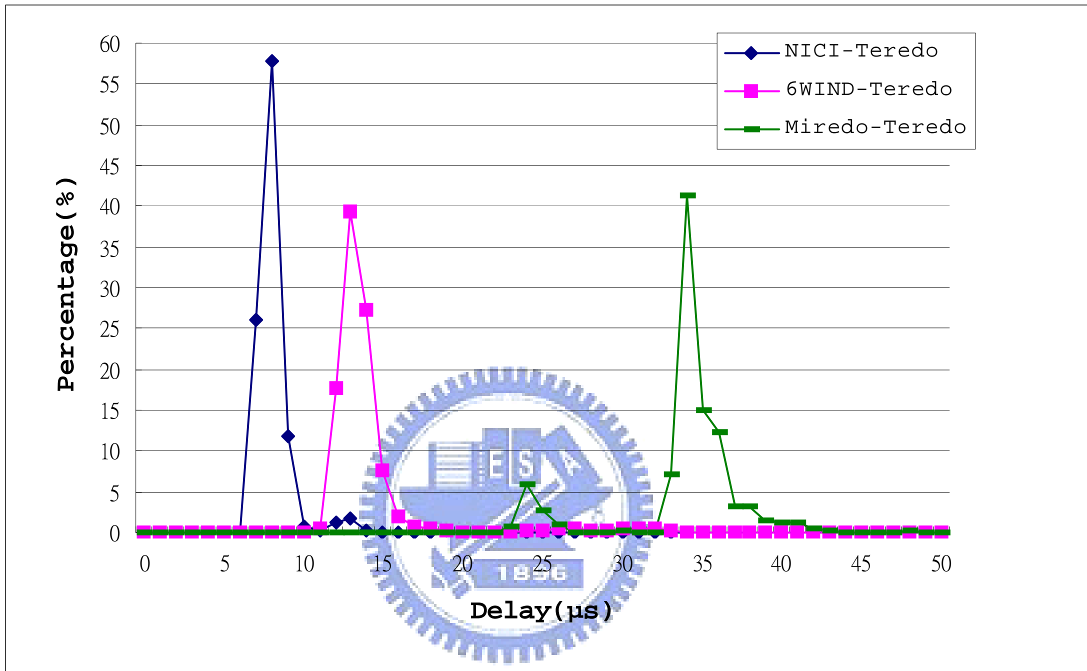


Figure 10. Encapsulation latency histograms of the Teredo relays (1280 bytes)

Table 1. Average processing latency

Teredo relay	Avg. latency of encapsulation (μs)			Avg. latency of decapsulation (μs)		
	64 bytes	512 bytes	1280 bytes	64 bytes	512 bytes	1280 bytes
NICI	7.69	7.82	8.06	8.77	9.10	9.24
6WIND	13.30	13.53	14.00	14.34	14.62	14.80
Miredo	23.05	25.46	33.08	24.90	33.14	34.85

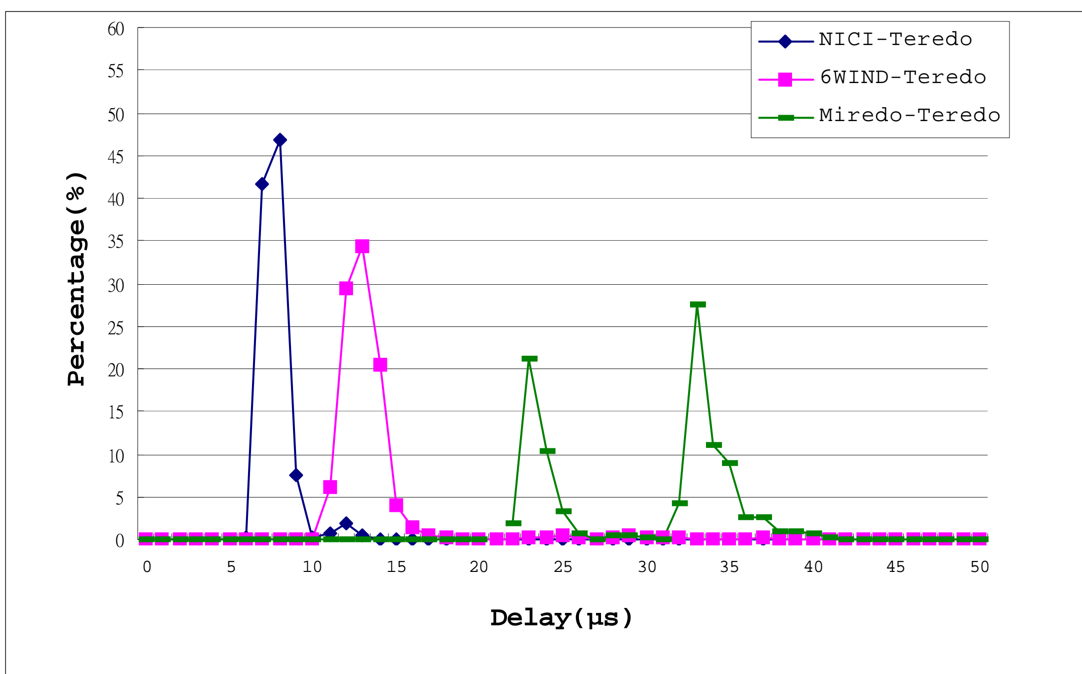


Figure 11. Encapsulation latency histograms of the Teredo relays (512 bytes)

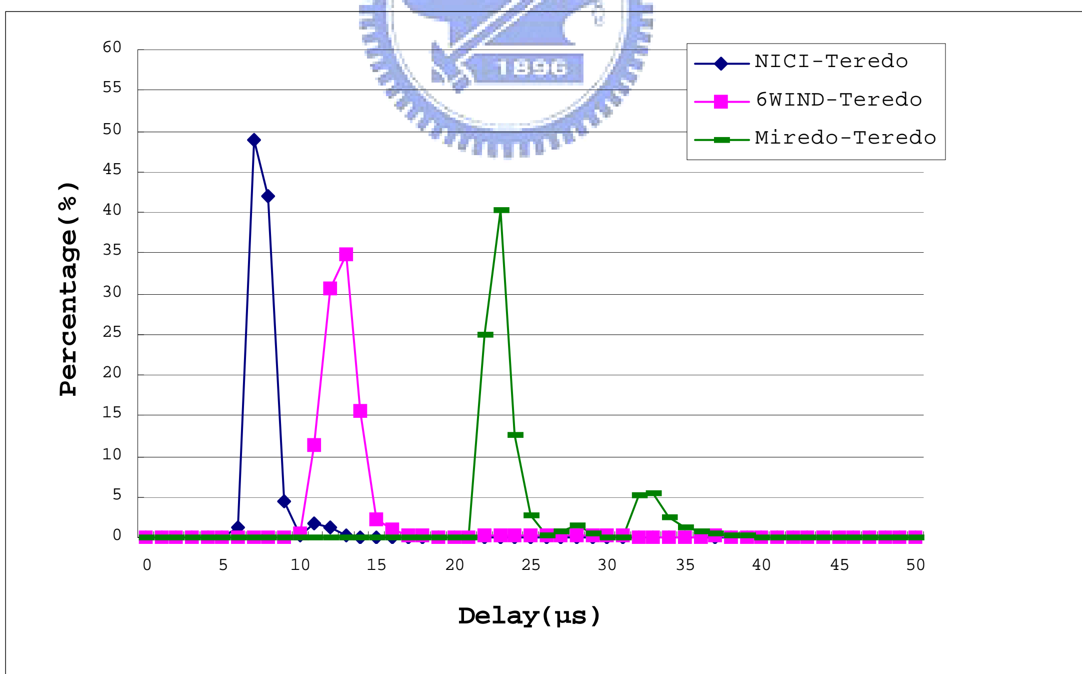
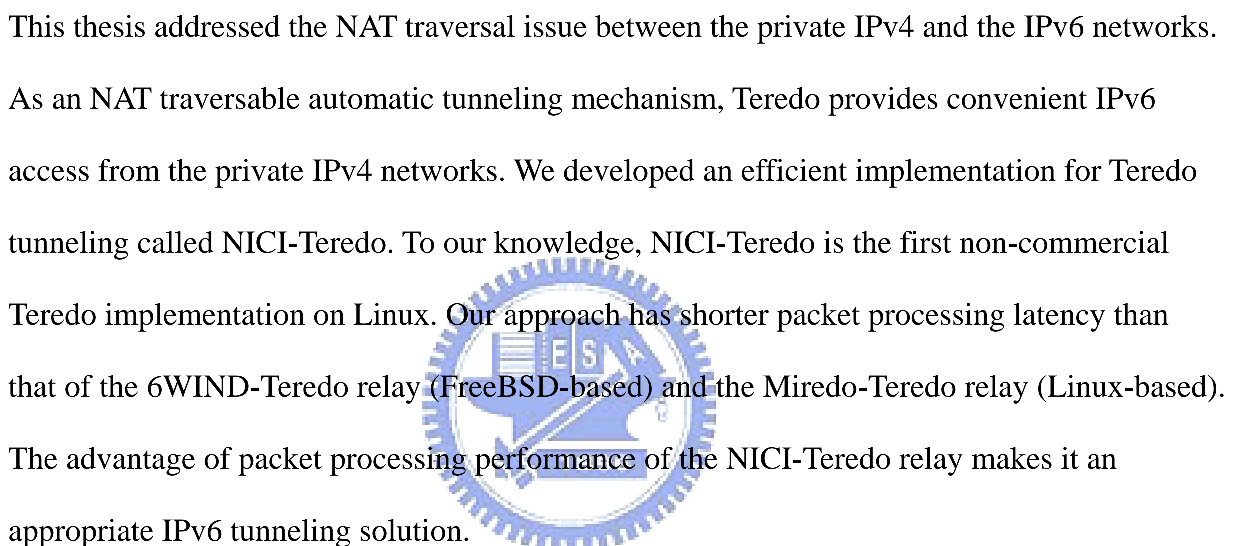


Figure 12. Encapsulation latency histograms of the Teredo relays (64 bytes)

With different packet sizes (1280, 512, 64 bytes), the encapsulation latency histograms of NICI-Teredo relay and 6WIND-Teredo relay (see Figures 10, 11 and 12) have similar shapes, where the Miredo-Teredo relay has longer latency than that of the NICI-Teredo relay. The decapsulation latency histograms of the three Teredo relays can be found in Appendix A. The histograms in Appendix A shows again that NICI-Teredo relay has the best packet processing performance among the three implementations of Teredo relays. For the 1280-byte packet encapsulation latency listed in Table 1, the latency improvement of the NICI-Teredo relay over the 6WIND-Teredo relay is around 42%, and the improvement of the NICI-Teredo relay over the Miredo-Teredo relay is around 75%.

Chapter 5
Conclusions

This thesis addressed the NAT traversal issue between the private IPv4 and the IPv6 networks. As an NAT traversable automatic tunneling mechanism, Teredo provides convenient IPv6 access from the private IPv4 networks. We developed an efficient implementation for Teredo tunneling called NICI-Teredo. To our knowledge, NICI-Teredo is the first non-commercial Teredo implementation on Linux. Our approach has shorter packet processing latency than that of the 6WIND-Teredo relay (FreeBSD-based) and the Miredo-Teredo relay (Linux-based). The advantage of packet processing performance of the NICI-Teredo relay makes it an appropriate IPv6 tunneling solution.



As a final remark, we point out that Teredo does not work for all NAT servers. According to the rules for port mapping and access control in NAT [9], the four major types are full cone NAT, restricted cone NAT, port restricted cone NAT, and symmetric NAT. Although Teredo can successfully traverse the first three types of NATs, it fails in traversing symmetric NAT. To enhance Teredo for symmetric NAT is still an open issue for further study.

Bibliography

[1] R. Gilligan and E. Nordmark, “Transition Mechanisms for IPv6 Hosts and Routers”, IETF RFC 2893, Aug. 2000.

[2] B. Carpenter and K. Moore, “Connection of IPv6 Domains via IPv4 Clouds”, IETF RFC 3056, Feb. 2001.

[3] A. Durand, P. Fasano, I. Guardini and D. Lento, “IPv6 Tunnel Broker”, IETF RFC 3053, Jan. 2001.

[4] Y.-B. Lin and I. Chlamtac, Wireless and Mobile Network Architectures. John Wiley & Sons, 2001.

[5] P. Srisuresh and M. Holdrege, “IP Network Address Translator (NAT) Terminology and Considerations”, IETF RFC 2663, Aug. 1999.

[6] H. Levkowetz and S. Vaarala, “Mobile IP Traversal of Network Address Translation (NAT) Devices”, IETF RFC 3519, Apr. 2003.

[7] C. Huitema, “Teredo: Tunneling IPv6 over UDP through NATs”, Internet draft, draft-huitema-v6ops-teredo-05.txt (Work in Progress), Apr. 2005.

[8] M. Liu, X. Wu, Y. Cai, M. Jin and D. Li, “Tunneling IPv6 with private IPv4 addresses through NAT devices”, Internet Draft, draft-liumin-v6ops-silkroad-02.txt (Work in Progress), Nov. 2004.

[9] J. Rosenberg, J. Weinberger, C. Huitema and R. Mahy, “STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs)”, IETF RFC 3489, Mar. 2003.

[10] S.-M. Huang and Q. Wu, “Implementation of Teredo - Tunneling IPv6 through NATs”, Technical Report for National Information and Communication Initiative (NICI) IPv6 R&D Division, Taiwan, ROC, 2003.

[11] Teredo for FreeBSD, http://www-rp.lip6.fr/teredo/

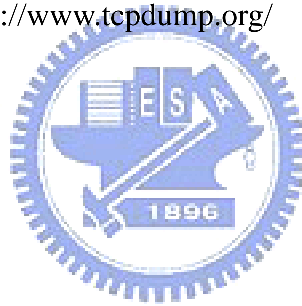
[12] Miredo: Teredo for Linux, http://www.simphalempin.com/dev/miredo/

[13] B. Henderson, Linux Loadable Kernel Module HOWTO, http://www.linux.org/docs/ldp/howto/Module-HOWTO/

[14] S. Bradner and J. McQuaid, “Benchmarking Methodology for Network Interconnect Devices”, IETF RFC 2544, Mar. 1999.

[15] The netfilter/iptables project, http://www.netfilter.org/

[16] Tcpdump and libpcap, http://www.tcpdump.org/



Appendix A
Decapsulation Latency Histograms

Appedix A is a supplement to the experimantal result in Chapter 4.2. This appendix shows the packet decapsulation latency histograms of the three Teredo relays (NICI-Teredo, 6WIND-Teredo and Miredo-Teredo). The 1280-byte result is showed in Figure 13, and the 512-byte and 64-byte results are showed in Figure 14 and 15,respectively.

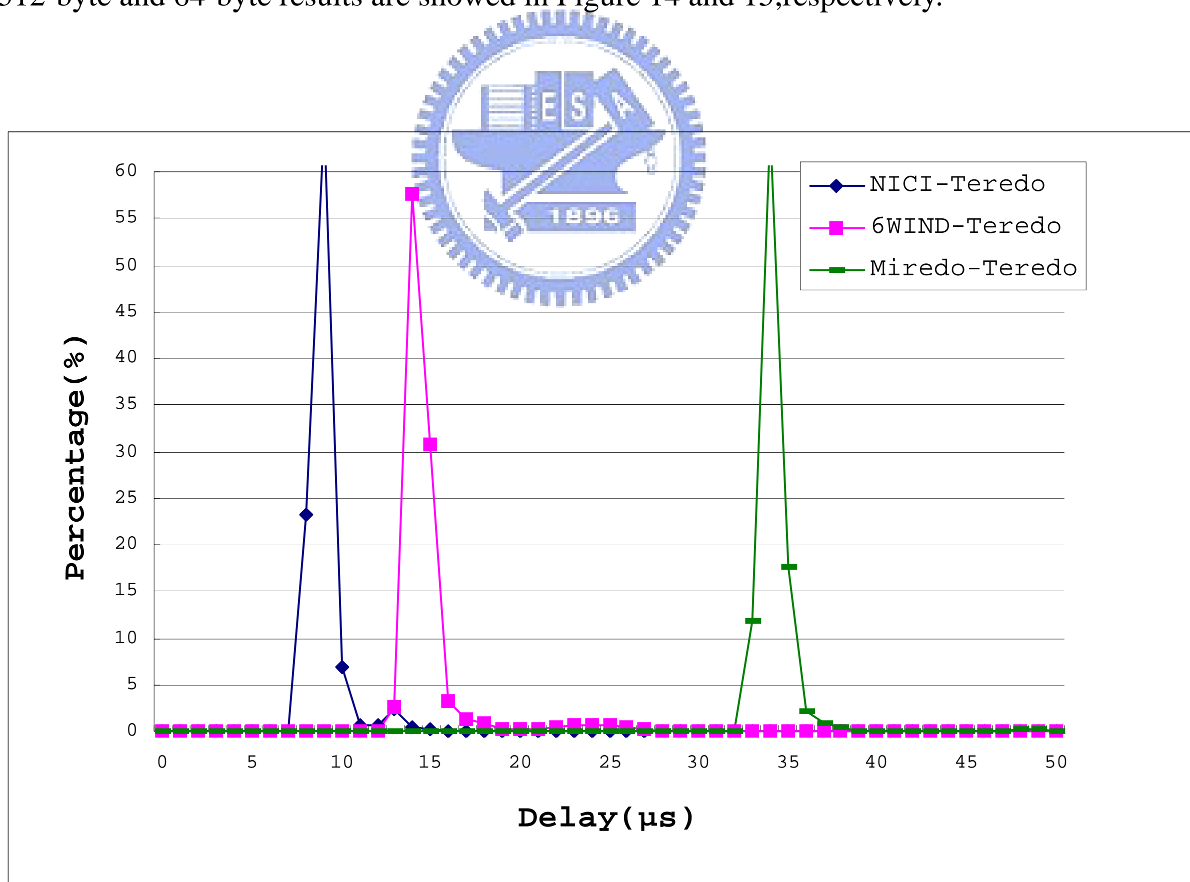


Figure 13. Decapsulation latency histograms of the Teredo relays (1280 bytes)

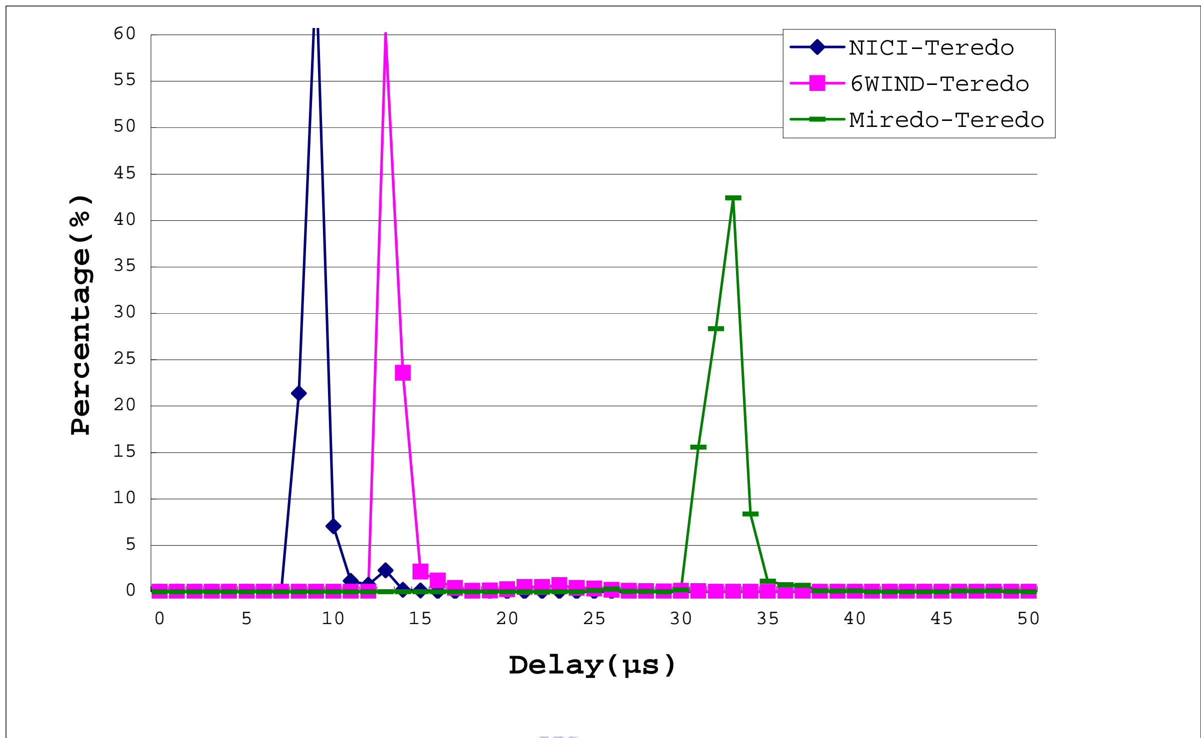


Figure 14. Decapsulation latency histograms of the Teredo relays (512 bytes)

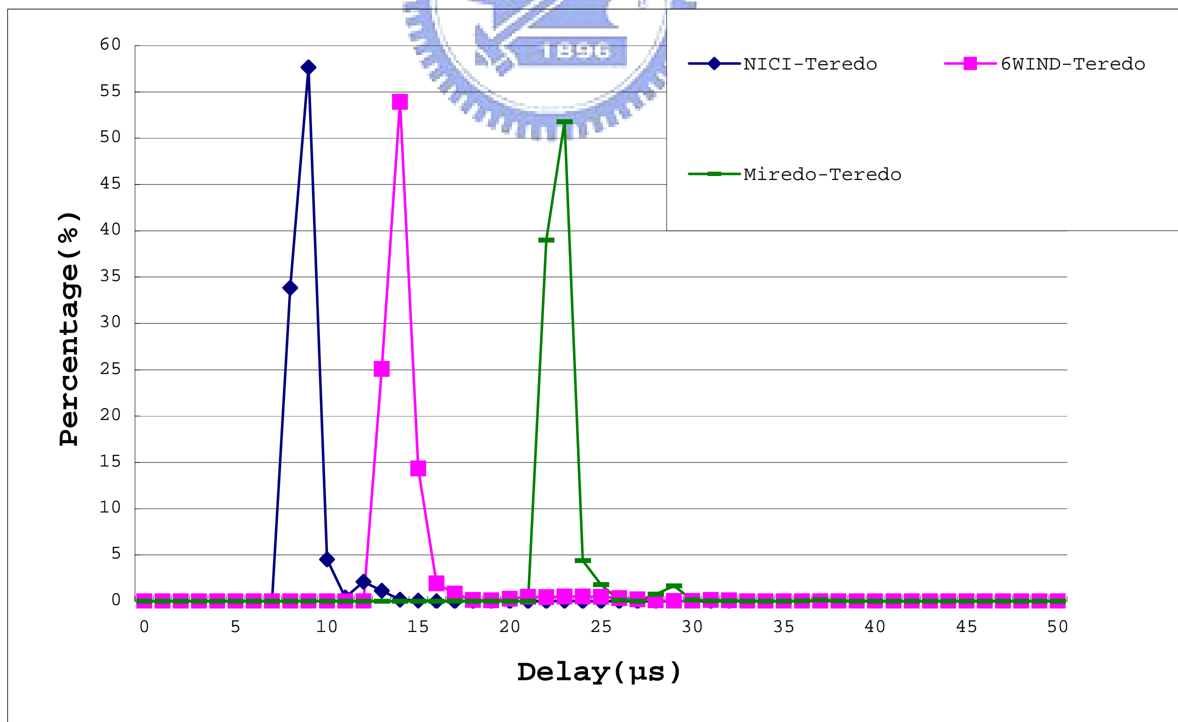
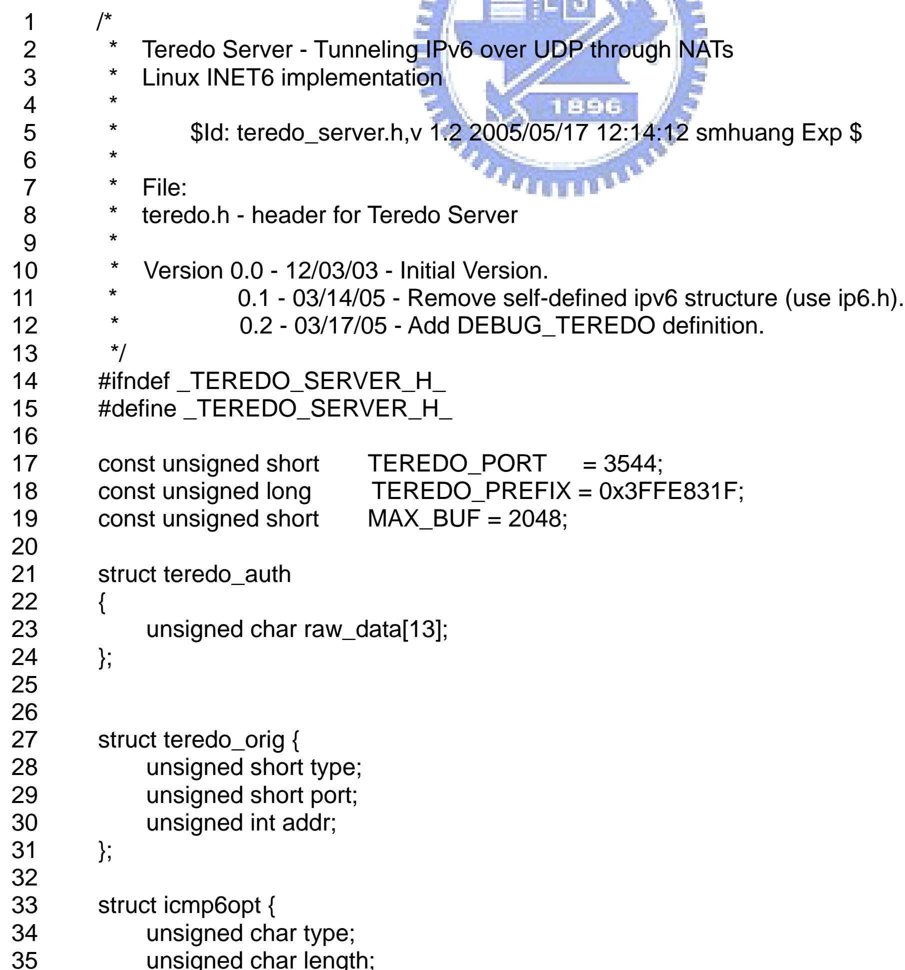


Figure 15. Decapsulation latency histograms of the Teredo relays (64 bytes)

Appendix B
The NICI-Teredo Server Program

Appendix B lists the source code of the NICI-Teredo server. The NICI-Teredo server is implemented as a C program, which consists of a header file **`teredo_server.h`** (Appendix B.1) and a program file **`teredo_server.c`** (Appendix B.2).

B.1 `teredo_server.h`



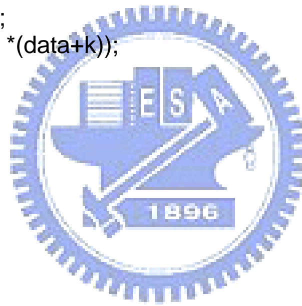
```
1    /*
2     *  Teredo Server - Tunneling IPv6 over UDP through NATs
3     *  Linux INET6 implementation
4     *
5     *      $Id: teredo_server.h,v 1.2 2005/05/17 12:14:12 smhuang Exp $
6     *
7     *  File:
8     *  teredo.h - header for Teredo Server
9     *
10    *  Version 0.0 - 12/03/03 - Initial Version.
11    *          0.1 - 03/14/05 - Remove self-defined ipv6 structure (use ip6.h).
12    *          0.2 - 03/17/05 - Add DEBUG_TEREDO definition.
13    */
14   #ifndef _TEREDO_SERVER_H_
15   #define _TEREDO_SERVER_H_
16
17   const unsigned short     TEREDO_PORT   = 3544;
18   const unsigned long      TEREDO_PREFIX = 0x3FFE831F;
19   const unsigned short     MAX_BUF = 2048;
20
21   struct teredo_auth
22   {
23       unsigned char raw_data[13];
24   };
25
26
27   struct teredo_orig {
28       unsigned short type;
29       unsigned short port;
30       unsigned int addr;
31   };
32
33   struct icmp6opt {
34       unsigned char type;
35       unsigned char length;
```

```
36          unsigned char prefix_len;
37          unsigned char flag;
38          unsigned int valid_time;
39          unsigned int pre_time;
40          unsigned int reserved;
41          unsigned int prefix[4];
42      };
43
44      #ifdef DEBUG_TEREDO
45      #define DEBUG_STR(x)            x
46      #define DEBUG_TRACE(x)          printf(DEBUG_STR(x))
47      #define DEBUG_TRACE2(n,x)       if ( (n) > DEBUG_LEVEL ) { printf(DEBUG_STR(x)); };
48      #else
49      #define DEBUG_TRACE(x)
50      #define DEBUG_TRACE2(n,x)
51      #endif
52
53      #define ERROR_TRACE(x) printk(KERN_ERR x)
54
55
56      #endif
```

B.2 teredo_server.c

```
 1      /*
 2       *      Teredo Server - Tunneling IPv6 over UDP through NATs
 3       *      Linux INET6 implementation
 4       *
 5       *      File:
 6       *      teredo_server.c - Teredo Server main program
 7       *
 8       *   Version 0.0 - 12/03/03 - Initial Version.
 9       *              0.1 - 05/02/04 - Fix relay -> server bubble format.
10       *              0.2 - 08/21/04 - Fix checksum() algorithm.
11       *              0.3 - 03/14/05 - Fix bubble transmission function.
12       *              0.4 - 03/17/05 - Add packet format check.
13       */
14      #include <stdio.h>
15      #include <sys/socket.h>
16      #include <arpa/inet.h>
17
18      #include "teredo_server.h"
19      #include "in6.h"
20      #include "ipv6.h"
21      #include <linux/icmpv6.h>
22
23      /* server address */
24      unsigned long TEREDO_PRI_SERVER;
25      unsigned long TEREDO_SEC_SERVER;
26
27      int
28      isglobal (unsigned int addr)
29      {
30          return (addr&htonl(0xff000000) == 0 ||                  /* the "local" subnet 0.0.0.0/8 */
31                  addr&htonl(0xff000000) == htonl(0x7f000000) || /* the "loopback" subnet 127.0.0.0/8 */
32                  addr&htonl(0xff000000) == htonl(0x0a000000) || /* the local addressing ranges 10.0.0.0/8 */
33                  addr&htonl(0xfff00000) == htonl(0xac100000) || /* the local addressing ranges
172.16.0.0/12 */
34                  addr&htonl(0xffff0000) == htonl(0xc0a80000) || /* the local addressing ranges
192.168.0.0/16 */
35                  addr&htonl(0xffff0000) == htonl(0xa9fe0000) || /* the link local block 169.254.0.0/16 */
36                  addr&htonl(0xffffff00) == htonl(0xc0586300) || /* the block reserved for 6to4 anycast
```

```
addresses 192.88.99.0/24 */
 37                    addr&htonl(0xf0000000) == htonl(0xe0000000) || /* the multicast address block 224.0.0.0/4
*/
 38                    addr == 0xffffffff)?0:1;                    /* the "limited broadcast" destination
address 255.255.255.255 */
 39                    /* TODO: the directed broadcast addresses corresponding to the subnets */
 40        }
 41
 42        unsigned short
 43        checksum(unsigned short v6_payload_len, unsigned short v6_nexthdr,
 44                 struct in6_addr *v6_saddr, struct in6_addr *v6_daddr, unsigned short *data, int len)
 45        {
 46            unsigned short cksum = v6_payload_len + v6_nexthdr;
 47            int k;
 48            for (k=0; k<8; k++)
 49            {
 50                cksum += v6_saddr->in6_u.u6_addr16[k];
 51                cksum += (cksum < v6_saddr->in6_u.u6_addr16[k]);
 52                cksum += v6_daddr->in6_u.u6_addr16[k];
 53                cksum += (cksum < v6_daddr->in6_u.u6_addr16[k]);
 54            }
 55            for (k=0; k<len; k++)
 56            {
 57                if (k == 1)
 58                {
 59                    continue;
 60                }
 61                cksum += *(data+k);
 62                cksum += (cksum < *(data+k));
 63            }
 64            cksum ^= 0xffff;
 65
 66            return cksum;
 67        }
 68
 69
 70        int
 71        main (int argc, char* argv[])
 72        {
 73            int sockd, secfd, fd6;
 74            struct sockaddr_in    my_addr, my2_addr, peer_addr;
 75            struct sockaddr_in6 dest;
 76            const int on = 1;
 77            int status, addrlen = sizeof(peer_addr), fdflag;
 78
 79            unsigned char recvbuf[MAX_BUF], sendbuf[MAX_BUF];
 80
 81            struct teredo_orig* torig;
 82            struct teredo_auth* tauth;
 83            struct ipv6hdr* rv6hdr, *sv6hdr;
 84            struct icmp6hdr* mp6hdr;
 85            struct icmp6opt* mp6opt;
 86
 87            fd_set fdSet;
 88            int fdSetSize;
 89
 90            if (argc != 3)
 91            {
 92                printf("Usage: %s PRI_SERVER_IPv4_ADDR SEC_SERVER_IPv4_ADDR\n", argv[0]);
 93                exit(1);
 94            }
 95
 96            /* a daemon */
 97            pid_t pid = fork();
 98            if (pid < 0)
 99            {
```



```
100                 perror("fork");
101                 exit(1);
102             }
103             else if (pid > 0) /* parent exit */
104             {
105                 exit(0);
106             }
107             setsid();
108             fclose(stdin); fclose(stdout);
109
110             if (inet_pton(AF_INET, argv[1],   &TEREDO_PRI_SERVER) <= 0 ||
111                 inet_pton(AF_INET, argv[2],   &TEREDO_SEC_SERVER) <= 0 )
112             {
113                 perror("inet_pton");
114                 exit(1);
115             }
116
117             /*   UDP socket for primary server */
118             sockd = socket(PF_INET, SOCK_DGRAM, 0);
119             if (sockd == -1)
120             {
121                     perror("socket");
122                     exit(1);
123             }
124             my_addr.sin_family = AF_INET;
125             my_addr.sin_addr.s_addr = TEREDO_PRI_SERVER;
126             my_addr.sin_port = htons(TEREDO_PORT);
127             status = bind(sockd, (struct sockaddr*)&my_addr, sizeof(my_addr));
128             if (status   == -1)
129             {
130                     perror("bind");
131                     exit(1);
132             }
133
134             /* UDP socket for secondary server */
135             secfd = socket(PF_INET, SOCK_DGRAM, 0);
136             if (secfd   == -1)
137             {
138                     perror("socket");
139                     exit(1);
140             }
141             my2_addr.sin_family = AF_INET;
142             my2_addr.sin_addr.s_addr = TEREDO_SEC_SERVER;
143             my2_addr.sin_port = htons(TEREDO_PORT);
144             status = bind(secfd, (struct sockaddr*)&my2_addr, sizeof(my2_addr));
145             if (status   == -1)
146             {
147                     perror("bind");
148                     exit(1);
149             }
150
151             /* IPv6 raw socket */
152             fd6 = socket(PF_INET6, SOCK_RAW, IPPROTO_RAW);
153             if (fd6   == -1)
154             {
155                     perror("socket");
156                     exit(1);
157             }
158
159             for (;;)
160             {
161                     status = 0;
162
163                     FD_ZERO(&fdSet);
164                     FD_SET(sockd, &fdSet);
165                     FD_SET(secfd, &fdSet);
```

```
166
167                     if (sockd > secfd)
168                     {
169                         fdSetSize = sockd + 1;
170                     }
171                     else
172                     {
173                         fdSetSize = secfd + 1;
174                     }
175
176                     select(fdSetSize, &fdSet, NULL, NULL, NULL);
177
178                     if (FD_ISSET(sockd, &fdSet))
179                     {
180                         fdflag = 0;
181                         status = recvfrom(sockd, recvbuf, MAX_BUF, 0, (struct sockaddr*)&peer_addr,
&addrlen);
182                         fprintf(stderr, "recvfrom sockd: %d\n", status);
183                     }
184                     if (FD_ISSET(secfd, &fdSet))
185                     {
186                         fdflag = 1;
187                         status = recvfrom(secfd, recvbuf, MAX_BUF, 0, (struct sockaddr*)&peer_addr,
&addrlen);
188                         fprintf(stderr, "recvfrom secfd: %d\n", status);
189                     }
190                     /* check packet format according to the draft... */
191                 /*
192                 1) If the UDP content is not a well formed Teredo IPv6 packet, as
193                 defined in section 5.1.1, the packet MUST be silently discarded.
194                 */
195                 if (status < sizeof(struct ipv6hdr))
196                 {
197                     continue;
198                 }
199                 if (recvbuf[0] != 0x60 && recvbuf[0] != 0x00)
200                 {
201                     continue;
202                 }
203                 /*
204                 2) If the UDP packet is not a Teredo bubble or an ICMPv6 message, it
205                 SHOULD be discarded. (The packet may be processed if the Teredo
206                 server also operates as a Teredo relay, as explained in section
207                 5.4.)
208                 */
209                 if (status == sizeof(struct ipv6hdr)) // bubble
210                 {
211                     rv6hdr = (struct ipv6hdr*) recvbuf;
212                     if (rv6hdr->version != 6 || rv6hdr->nexthdr != IPPROTO_NONE || rv6hdr->payload_len != 0 )
213                     {
214                         continue;
215                     }
216                 }
217                 else if (recvbuf[0] == 0x00)
218                 {
219                     if (status < sizeof(struct teredo_orig)+sizeof(struct ipv6hdr))
220                     {
221                         continue;
222                     }
223                     rv6hdr = (struct ipv6hdr*) &recvbuf[sizeof(struct teredo_auth)];
224                     if (rv6hdr->nexthdr != IPPROTO_ICMPV6)
225                     {
226                         continue;
227                     }
228                 }
229                 else if (recvbuf[0] == 0x60)
```

```
230                 {
231                     rv6hdr = (struct ipv6hdr*) recvbuf;
232                     if (rv6hdr->nexthdr != IPPROTO_ICMPV6)
233                     {
234                         continue;
235                     }
236                 }
237                 /*
238                 3) If the IPv4 source address is not in the format of a global
239                 unicast address, the packet MUST be silently discarded (see
240                 section 5.2.4 for a definition of global unicast addresses).
241                 */
242                 if (!isglobal(peer_addr.sin_addr.s_addr))
243                 {
244                     continue;
245                 }
246                 /*
247                 If the destination IPv6 address is not a global scope IPv6 address,
248                 the packet MUST NOT be forwarded. ---see RFC 2373
249                 */
250                 if (rv6hdr->daddr.in6_u.u6_addr16[0]&htons(0xffc0) == htons(0xff80) ||
251                     rv6hdr->daddr.in6_u.u6_addr16[0]&htons(0xffc0) == htons(0xfe80) ||
252                     rv6hdr->daddr.in6_u.u6_addr16[0]&htons(0xffc0) == htons(0xfec0) )
253                 {
254                     continue;
255                 }
256                 /*
257                 If the destination IPv6 address is a Teredo client whose address is
258                 serviced by this specific server, the server should insert an origin
259                 indication in the first bytes of the UDP payload,
260                 */
261                 if (rv6hdr->daddr.in6_u.u6_addr32[0] == htonl(0x3ffe831f))
262                 {
263                     if (rv6hdr->daddr.in6_u.u6_addr32[1] == TEREDO_PRI_SERVER)
264                     {
265                         if (!isglobal(rv6hdr->daddr.in6_u.u6_addr32[3]))
266                         {
267                             continue;
268                         }
269                     }
270                     else
271                     {
272                         continue;
273                     }
274                 }
275 
276                     if (status == sizeof(struct ipv6hdr)) /* receives a bubble, relay it */
277                     {
278                     fprintf(stderr, "source addr %x:%x\n", peer_addr.sin_addr, ntohs(peer_addr.sin_port));
279 
280                         /* add origin indication*/
281                         torig = (struct teredo_orig*) sendbuf;
282                     torig->type = 0;
283                         torig->port = peer_addr.sin_port^0xffff;
284                         torig->addr = peer_addr.sin_addr.s_addr^0xffffffff;
285 
286                         rv6hdr = (struct ipv6hdr*) recvbuf;
287                         peer_addr.sin_port = rv6hdr->daddr.in6_u.u6_addr16[5]^0xffff;
288                         peer_addr.sin_addr.s_addr = rv6hdr->daddr.in6_u.u6_addr32[3]^0xffffffff;
289                     peer_addr.sin_family = AF_INET;
290 
291                     memcpy(&sendbuf[sizeof(struct teredo_orig)], recvbuf, sizeof(struct ipv6hdr));
292 
293                     if (!isglobal(peer_addr.sin_addr.s_addr))
294                     {
295                         continue;
```

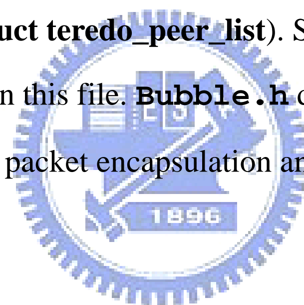
```
296                     }
297                             status = sendto(sockd, sendbuf, sizeof(struct teredo_orig)+sizeof(struct ipv6hdr), 0,
(struct sockaddr*)&peer_addr, sizeof(peer_addr));
298
299                             fprintf(stderr, "sendto(bubble): %d\n", status);
300                     }
301                     else if (status == sizeof(struct teredo_auth)+sizeof(struct ipv6hdr)+sizeof(struct
icmp6hdr)+16)
302                     { /* receives a router solicitation, (option size=16)*/
303                             /* add authentication indication */
304                     memcpy(sendbuf, recvbuf, sizeof(struct teredo_auth));
305
306                             /* add original indication */
307                             torig = (struct teredo_orig*) &sendbuf[sizeof(struct teredo_auth)];
308                             torig->type = 0;
309                             torig->port = peer_addr.sin_port^0xffff;
310                             torig->addr = peer_addr.sin_addr.s_addr^0xffffffff;
311
312                             /* add IPv6 header */
313                             rv6hdr = (struct ipv6hdr*) &recvbuf[sizeof(struct teredo_auth)];
314                             sv6hdr = (struct ipv6hdr*) &sendbuf[sizeof(struct teredo_auth)+sizeof(struct
teredo_orig)];
315                     sv6hdr->version = 6;
316                     sv6hdr->priority = IPV6_PRIORITY_UNCHARACTERIZED;
317                     sv6hdr->flow_lbl[0] = 0; sv6hdr->flow_lbl[1] = 0; sv6hdr->flow_lbl[2] = 0;
318                     sv6hdr->payload_len = htons(sizeof(struct icmp6hdr)+sizeof(struct icmp6opt)+8);
319                     sv6hdr->nexthdr = IPPROTO_ICMPV6; /* ICMPv6 */
320                     sv6hdr->hop_limit = 255;
321                     sv6hdr->saddr.in6_u.u6_addr32[0] = htonl(0xfe800000); sv6hdr->saddr.in6_u.u6_addr32[1]
= htonl(0x00000000);
322                     sv6hdr->saddr.in6_u.u6_addr16[4] = htons(0x8000);
323                     sv6hdr->saddr.in6_u.u6_addr16[5] = htons(TEREDO_PORT)^0xffff;
324                     sv6hdr->saddr.in6_u.u6_addr32[3] = TEREDO_PRI_SERVER^0xffffffff;
325
326                     sv6hdr->daddr.in6_u.u6_addr32[0] = rv6hdr->saddr.in6_u.u6_addr32[0];
327                     sv6hdr->daddr.in6_u.u6_addr32[1] = rv6hdr->saddr.in6_u.u6_addr32[1];
328                     sv6hdr->daddr.in6_u.u6_addr32[2] = rv6hdr->saddr.in6_u.u6_addr32[2];
329                     sv6hdr->daddr.in6_u.u6_addr32[3] = rv6hdr->saddr.in6_u.u6_addr32[3];
330                     ///////////////
331                             /*if (sv6hdr->daddr.in6_u.u6_addr8[8] == 0x80 && fdflag == 0)
332                             {
333                                     continue;
334                             }
335                             else if (fdflag == 0)
336                             {
337                                     torig->port = htons(TEREDO_PORT)^0xffff;
338                             }
339                             else
340                             {
341                                     torig->port = htons(7788)^0xffff;
342                             }*/
343                             /////////////SMTEST
344
345                             /* ICMPv6 header */
346                             mp6hdr = (struct icmp6hdr*) &sendbuf[sizeof(struct teredo_auth)+sizeof(struct
teredo_orig)+sizeof(struct ipv6hdr)];
347                     mp6hdr->icmp6_type = 134; /* router advertisement */
348                     mp6hdr->icmp6_code = 0;
349                     mp6hdr->icmp6_cksum = 0; /* calculate later */
350                     mp6hdr->icmp6_dataun.un_data32[0]= 0;
351
352                     int i = sizeof(struct teredo_auth)+sizeof(struct teredo_orig)+sizeof(struct
ipv6hdr)+sizeof(struct icmp6hdr);
353                     sendbuf[i++] = 0x00; sendbuf[i++] = 0x00; sendbuf[i++] = 0x00; sendbuf[i++] = 0x00;
354                     sendbuf[i++] = 0x00; sendbuf[i++] = 0x00; sendbuf[i++] = 0x07; sendbuf[i++] = 0xd0;
355
```

```
356                         /* ICMPv6 options */
357                         mp6opt = (struct icmp6opt*) &sendbuf[i];
358                         mp6opt->type = 3;
359                         mp6opt->length = 4;
360                         mp6opt->prefix_len = 64;
361                         mp6opt->flag = 0x40;
362                         mp6opt->valid_time = 0xffffffff;
363                         mp6opt->pre_time = 0xffffffff;
364                         mp6opt->reserved = 0;
365                         mp6opt->prefix[0] = htonl(TEREDO_PREFIX);
366                         mp6opt->prefix[1] = TEREDO_PRI_SERVER;
367                         mp6opt->prefix[2] = 0;
368                         mp6opt->prefix[3] = 0;
369
370                     mp6hdr->icmp6_cksum = checksum(sv6hdr->payload_len, htons(sv6hdr->nexthdr),
371                                                 &(sv6hdr->saddr), &(sv6hdr->daddr),
372                                                 (unsigned short*)&sendbuf[sizeof(struct
teredo_auth)+sizeof(struct teredo_orig)+sizeof(struct ipv6hdr)],
373                                                 (int) ntohs(sv6hdr->payload_len)/2);
374
375                     if ( fdflag == 0)
376                     {
377                         if (sv6hdr->daddr.in6_u.u6_addr8[8] == 0x80)
378                             {
379                             if (!isglobal(peer_addr.sin_addr.s_addr))
380                             {
381                                 continue;
382                             }
383                                     status = sendto(secfd, sendbuf, i+sizeof(struct icmp6opt), 0, (struct
sockaddr*)&peer_addr, sizeof(peer_addr));
384                                 fprintf(stderr, "sendto(secfd): %d\n", status);
385                             }
386                             else
387                             {
388                             if (!isglobal(peer_addr.sin_addr.s_addr))
389                             {
390                                 continue;
391                             }
392                                     status = sendto(sockd, sendbuf, i+sizeof(struct icmp6opt), 0, (struct
sockaddr*)&peer_addr, sizeof(peer_addr));
393                                 fprintf(stderr, "sendto(sockd): %d\n", status);
394                             }
395                         }
396                         else
397                         {
398                         if (sv6hdr->daddr.in6_u.u6_addr8[8] == 0x80)
399                             {
400                             if (!isglobal(peer_addr.sin_addr.s_addr))
401                             {
402                                 continue;
403                             }
404                                     status = sendto(sockd, sendbuf, i+sizeof(struct icmp6opt), 0, (struct
sockaddr*)&peer_addr, sizeof(peer_addr));
405                                 fprintf(stderr, "sendto(sockd): %d\n", status);
406                             }
407                             else
408                             {
409                             if (!isglobal(peer_addr.sin_addr.s_addr))
410                             {
411                                 continue;
412                             }
413                                     status = sendto(secfd, sendbuf, i+sizeof(struct icmp6opt), 0, (struct
sockaddr*)&peer_addr, sizeof(peer_addr));
414                                 fprintf(stderr, "sendto(secfd): %d\n", status);
415                             }
416                         }
```

```
417                     }
418                     else if (rv6hdr->saddr.in6_u.u6_addr32[0] == htonl(0x3ffe831f) &&
419                             rv6hdr->daddr.in6_u.u6_addr32[0] == htonl(0x3ffe831f) )
420                     { /* to another Teredo client */
421                         peer_addr.sin_port = rv6hdr->daddr.in6_u.u6_addr16[5]^0xffff;
422                         peer_addr.sin_addr.s_addr = rv6hdr->daddr.in6_u.u6_addr32[3]^0xffffffff;
423                     peer_addr.sin_family = AF_INET;
424
425                     memcpy(&sendbuf[sizeof(struct teredo_orig)], recvbuf, sizeof(struct ipv6hdr));
426
427                     if (!isglobal(peer_addr.sin_addr.s_addr))
428                     {
429                         continue;
430                     }
431                         status = sendto(sockd, sendbuf, sizeof(struct teredo_orig)+sizeof(struct ipv6hdr), 0,
(struct sockaddr*)&peer_addr, sizeof(peer_addr));
432
433                         fprintf(stderr, "sendto(client): %d\n", status);
434                     }
435                     else /* receives a ping6 packet from a Teredo client */
436                     {
437                         dest.sin6_family = AF_INET6;
438                         dest.sin6_addr.s6_addr16[0] = 1; dest.sin6_addr.s6_addr16[1] = 1;
dest.sin6_addr.s6_addr16[2] = 1;
439                         dest.sin6_addr.s6_addr16[3] = 1; dest.sin6_addr.s6_addr16[4] = 1;
dest.sin6_addr.s6_addr16[5] = 1;
440                         dest.sin6_addr.s6_addr16[6] = 1; dest.sin6_addr.s6_addr16[7] = htons(1);
441                         dest.sin6_flowinfo = 0;
442                         dest.sin6_scope_id = 0;
443                         dest.sin6_port = 0;
444
445                     if (!isglobal(peer_addr.sin_addr.s_addr))
446                     {
447                         continue;
448                     }
449                         status = sendto(fd6, recvbuf, status, 0, (struct sockaddr*)&dest, sizeof(dest));
450                         fprintf(stderr, "sendto6: %d\n", status);
451                     }
452             } /* end for loop */
453
454             return 0;
455         }
456
```

Appendix C
The NICI-Teredo Relay Program

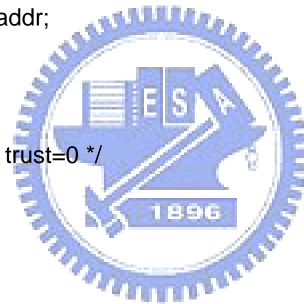
Appendix C lists the source code of the NICI-Teredo relay. The NICI-Teredo relay is implemented as a C program, which consists of two header files **`teredo.h`** (Appendix C.1), **`bubble.h`** (Appendix C.2) and a program file **`teredo.c`** (Appendix C.3). In **`teredo.h`**, a data structure for maintenance the tunnel reachability between the Teredo client and the Teredo relay is defined (i.e. **struct teredo_peer_list**). Several data structures for kernel level configuration are also defined in this file. **`Bubble.h`** defince several inline functions for handling packet tunneling. The packet encapsulation and decapsulation functions are defined in **`teredo.c`**.



C.1 `teredo.h`

```
 1   /*
 2    *  Teredo tunnel device - Tunneling IPv6 over UDP through NATs
 3    *  Linux INET6 implementation
 4    *
 5    *
 6    *       $Id: teredo.h,v 1.13 2005/03/10 12:50:03 smhuang Exp $
 7    *
 8    *  This program is free software; you can redistribute it and/or
 9    *  modify it under the terms of the GNU General Public License
10    *  as published by the Free Software Foundation; either version
11    *  2 of the License, or (at your option) any later version.
12    *
13    *  Version 0.0 - 10/24/03 - Initial Version.
14    *              0.1 - 12/24/03 - Add Teredo Bubble support.
15    *              0.2 - 01/12/04 - Bug Fix (tunnel packet format in udpip6_tunnel_xmit).
16    *              0.3 - 03/16/04 - Add kernel thread for Bubble retransmission.
17    *              0.4 - 01/15/05 - Remove kernel thread feature.
18    *              0.5 - 03/11/05 - Add packet check feature.
19    */
20
21
22   #ifndef _TEREDO_H_
```

```
23	#define _TEREDO_H_
24
25
26	#define __NO_VERSION__
27	#include <linux/kernel.h>
28
29
30	#define TEREDO_MTU 1280
31
32	#define TEREDO_PORT 3544
33
34	#define TEREDO_MAX_PEER		0xFF
35	#define TEREDO_QLEN	1
36
37	#define TEREDO_TOUT 30	/* 30 seconds */
38
39
40	struct teredo_peer_list {
41	/*
42	     - The IPv6 address of the peer,
43	     - The mapped IPv4 address and mapped UDP port of the peer,
44	     - The status of the mapped address, i.e. trusted or not,
45	     - The value of the last "nonce" sent to the peer,
46	     - The date and time of the last reception from the peer,
47	     - The date and time of the last transmission to the peer,
48	     - The number of bubbles transmitted to the peer.
49	*/
50		struct	in6_addr	v6addr;
51
52		u16 mapped_port;
53		u32 mapped_addr;
54
55		int status; /* turst=1, not trust=0 */
56		u64 nonce;
57
58		struct timeval last_rx;
59		struct timeval last_tx;
60
61		int tx_bubble;
62
63		//self define
64		//limit its size --ijk--
65		struct sk_buff_head pktq;
66	} teredo_peer[TEREDO_MAX_PEER+1];
67
68	static int udpip6_tunnel_init(struct net_device* dev);
69	int udpip6_rcv(struct sk_buff* skb);
70
71
72	static struct
73	net_device udpip6_tunnel_dev = {
74		name:		"teredo0",
75		init:		udpip6_tunnel_init,
76	};
77
78	static struct
79	ip_tunnel udpip6_tunnel = {
80		NULL, &udpip6_tunnel_dev, {0, }, 0, 0, 0, 0, 0, 0, 0, {"teredo0", }
81	};
82
83	static struct
84	inet_protocol teredo_protocol = {
85		udpip6_rcv,
86		0,
87		0,
88		IPPROTO_UDP,
```



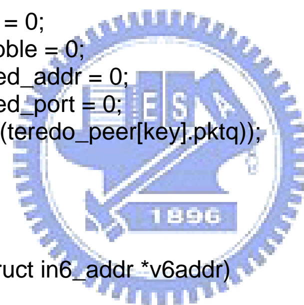
```
89          0,
90          NULL,
91          "IPv6"
92      };
93
94      static struct ip_tunnel *teredo_tunnel;
95
96      static rwlock_t udpip6_lock = RW_LOCK_UNLOCKED;
97
98      #ifdef DEBUG_TEREDO
99      #define DEBUG_STR(x)                KERN_DEBUG x
100     #define DEBUG_TRACE(x)               printk(DEBUG_STR(x))
101     #define DEBUG_TRACE2(n,x)           if ( (n) > DEBUG_LEVEL ) { printk x; };
102     #else
103     #define DEBUG_TRACE(x)
104     #define DEBUG_TRACE2(n,x)
105     #endif
106
107     #define ERROR_TRACE(x) printk(KERN_ERR x)
108
109
110     #endif
```

C.2 bubble.h

```
1       /*
2        *   Teredo tunnel device - Tunneling IPv6 over UDP through NATs
3        *   Linux INET6 implementation
4        *
5        *
6        *        $Id: bubble.h,v 1.26 2005/03/10 12:49:36 smhuang Exp $
7        *
8        *   This program is free software; you can redistribute it and/or
9        *   modify it under the terms of the GNU General Public License
10       *   as published by the Free Software Foundation; either version
11       *   2 of the License, or (at your option) any later version.
12       *
13       *   Version 0.0 - 10/24/03 - Initial Version.
14       *               0.1 - 12/24/03 - Add Teredo Bubble support.
15       *               0.2 - 01/12/04 - Bug Fix (tunnel packet format in udpip6_tunnel_xmit).
16       *               0.3 - 03/16/04 - Add kernel thread for Bubble retransmission.
17       *               0.4 - 01/15/05 - Remove kernel thread feature.
18       *               0.5 - 03/11/05 - Add packet check feature.
19       */
20
21
22      #ifndef __BUBBLE_H_
23      #define __BUBBLE_H_
24
25      #include <linux/time.h>
26      #include "teredo.h"
27
28      /*
29       ******************************************************************
30       *   returns the embedded IPv4 addr. if the IPv6 addr.                *
31       *   comes from TEREDO(draft-huitema-v6ops-teredo-00) addr. space*
32       * +-------------+------------+-------+------+-------------+      *
33       * | Prefix         | Server IPv4 | Flags | Port | Client IPv4 |      *
34       * +-------------+------------+-------+------+-------------+      *
35       *   - Prefix: 32 bits Teredo service prefix.                          *
36       *   - Server IPv4: 32 bits IPv4 addr. of a Teredo server.             *
37       *   - Flags: 16 bits that document type of addr. and NAT.             *
```

```
38      *   - Port: the obfuscated "mapped UDP port" of the Teredo          *
39      *   service at the client.                                           *
40      *   - Client IPv4: the obfuscated "mapped IPv4 address" of a        *
41      *   client                                                                *
42      ****************************************************************
43      */
44     static inline u32
45     try_teredo(const struct in6_addr *v6dst)
46     {
47          u32 dst = 0;
48
49          if (v6dst->s6_addr16[0] == htons(0x3ffe) &&
50                 v6dst->s6_addr16[1] == htons(0x831f)     )
51          {
52               memcpy(&dst, &v6dst->s6_addr16[6], 4);
53               dst = dst ^ 0xffffffff;
54          }
55          return dst;
56     }
57
58     static inline u16
59     teredo_port(const struct in6_addr *v6dst)
60     {
61          u16 port = 0;
62          if (v6dst->s6_addr16[0] == htons(0x3ffe) &&
63               v6dst->s6_addr16[1] == htons(0x831f))
64          {
65               memcpy(&port, &v6dst->s6_addr16[5], 2);
66               port = port ^ 0xffff;
67          }
68          return port;
69     }
70
71     static inline u32
72     teredo_server_addr (const struct in6_addr *v6dst)
73     {
74          u32 dst = 0;
75          memcpy(&dst, &v6dst->s6_addr16[2], 4);
76          return dst;
77     }
78
79     static inline int
80     teredo_iscone (const struct in6_addr *v6dst)
81     {
82          return (v6dst->s6_addr16[4] == htons(0x8000))?1:0;
83     }
84
85     static inline int
86     teredo_isaddr (const struct in6_addr *v6dst)
87     {
88          return ((v6dst->s6_addr16[0] != htons(0x3ffe) || v6dst->s6_addr16[1] != htons(0x831f)))?0:1;
89          /*if (v6dst->s6_addr16[0] != htons(0x3ffe) ||
90               v6dst->s6_addr16[1] != htons(0x831f))
91          {
92               return 0;
93          }
94          else
95          {
96               return 1;
97          }*/
98     }
99
100    static inline u16
101    teredo_phash (const struct in6_addr *v6addr)
102    {
103         return
```

```
((v6addr->in6_u.u6_addr16[2])^(v6addr->in6_u.u6_addr16[3])^(v6addr->in6_u.u6_addr16[5])^
104                     (v6addr->in6_u.u6_addr16[6])^(v6addr->in6_u.u6_addr16[7]))&TEREDO_MAX_PEER;
105     }
106
107     static inline void
108     teredo_new_cone_peer (const struct in6_addr *v6addr)
109     {
110         u16 key = teredo_phash (v6addr);
111         teredo_peer[key].v6addr.in6_u.u6_addr32[0] = v6addr->in6_u.u6_addr32[0];
112         teredo_peer[key].v6addr.in6_u.u6_addr32[1] = v6addr->in6_u.u6_addr32[1];
113         teredo_peer[key].v6addr.in6_u.u6_addr32[2] = v6addr->in6_u.u6_addr32[2];
114         teredo_peer[key].v6addr.in6_u.u6_addr32[3] = v6addr->in6_u.u6_addr32[3];
115
116         teredo_peer[key].status = 1;
117         skb_queue_head_init(&(teredo_peer[key].pktq));
118         return;
119     }
120
121     static inline void
122     teredo_new_peer (const struct in6_addr *v6addr)
123     {
124         u16 key = teredo_phash (v6addr);
125         teredo_peer[key].v6addr.in6_u.u6_addr32[0] = v6addr->in6_u.u6_addr32[0];
126         teredo_peer[key].v6addr.in6_u.u6_addr32[1] = v6addr->in6_u.u6_addr32[1];
127         teredo_peer[key].v6addr.in6_u.u6_addr32[2] = v6addr->in6_u.u6_addr32[2];
128         teredo_peer[key].v6addr.in6_u.u6_addr32[3] = v6addr->in6_u.u6_addr32[3];
129
130         teredo_peer[key].status = 0;
131         teredo_peer[key].tx_bubble = 0;
132         teredo_peer[key].mapped_addr = 0;
133         teredo_peer[key].mapped_port = 0;
134         skb_queue_head_init(&(teredo_peer[key].pktq));
135         return;
136     }
137
138     static inline int
139     teredo_inc_bubble (const struct in6_addr *v6addr)
140     {
141         u16 key = teredo_phash (v6addr);
142         if (teredo_peer[key].tx_bubble <3)
143         {
144             teredo_peer[0].tx_bubble++;
145             return 0;
146         }
147         else
148         {
149             // remove the peer
150             return 1;
151         }
152     }
153
154     static inline int
155     teredo_ispeer (const struct in6_addr *v6addr)
156     {
157         u16 key = teredo_phash (v6addr);
158         if ((teredo_peer[key].v6addr.in6_u.u6_addr32[0] == v6addr->in6_u.u6_addr32[0]) &&
159             (teredo_peer[key].v6addr.in6_u.u6_addr32[1] == v6addr->in6_u.u6_addr32[1]) &&
160             (teredo_peer[key].v6addr.in6_u.u6_addr32[2] == v6addr->in6_u.u6_addr32[2]) &&
161             (teredo_peer[key].v6addr.in6_u.u6_addr32[3] == v6addr->in6_u.u6_addr32[3]) )
162         {
163             return 1;
164         }
165         else
166         {
167             return 0;
168         }
```



```
169     }
170
171     static inline int
172     teredo_istrust (const struct in6_addr *v6addr)
173     {
174         struct timeval tv;
175         u16 key = teredo_phash (v6addr);
176
177         do_gettimeofday(&tv);
178
179         if (teredo_peer[key].status == 0)
180         {
181             return 0;
182         }
183         else
184         {
185             //return ((tv.tv_sec- teredo_peer[key].last_tx.tv_sec) >= TEREDO_TOUT)?0:1;
186             return 1;
187         }
188     }
189
190     static inline void
191     teredo_trust (const struct in6_addr *v6addr)
192     {
193         u16 key = teredo_phash (v6addr);
194         teredo_peer[key].status = 1;
195         return;
196     }
197
198     static inline void
199     teredo_add_mapped(const struct in6_addr *v6addr, const u32 addr, const u16 port)
200     {
201         u16 key = teredo_phash (v6addr);
202         teredo_peer[key].mapped_addr = addr;
203         teredo_peer[key].mapped_port = port;
204         return;
205     }
206
207     static inline u16
208     teredo_peer_port(const struct in6_addr *v6addr)
209     {
210         u16 key = teredo_phash (v6addr);
211         return teredo_peer[key].mapped_port;
212     }
213
214     static inline u32
215     teredo_peer_addr(const struct in6_addr *v6addr)
216     {
217         u16 key = teredo_phash (v6addr);
218         return teredo_peer[key].mapped_addr;
219     }
220
221     static inline void
222     teredo_settime_tx (const struct in6_addr *v6addr)
223     {
224         u16 key = teredo_phash (v6addr);
225         do_gettimeofday(&(teredo_peer[key].last_tx));
226         return;
227     }
228
229     static inline void
230     teredo_settime_rx (const struct in6_addr *v6addr)
231     {
232         u16 key = teredo_phash (v6addr);
233         do_gettimeofday(&(teredo_peer[key].last_rx));
234         return;
```

```
235     }
236
237     static inline int
238     teredo_cmptime (const struct timeval *tva, const struct timeval *tvb)
239     {
240             return (tva->tv_sec - tva->tv_sec);
241     }
242
243     static inline void
244     teredo_build_bubble (struct sk_buff *skb)
245     {
246             struct ipv6hdr *iph6;
247
248             iph6 = skb->nh.ipv6h;
249             skb_trim(skb, sizeof(struct ipv6hdr));
250
251             iph6->priority = 0x0;
252             iph6->version = 0x6;
253             iph6->flow_lbl[0] = 0x0;
254             iph6->flow_lbl[1] = 0x0;
255             iph6->flow_lbl[2] = 0x0;
256             iph6->payload_len = 0x0;
257             iph6->nexthdr = IPPROTO_NONE;
258             iph6->hop_limit = 1;
259
260             return;
261     }
262
263     static inline void
264     teredo_enqueue (const struct in6_addr *v6addr, const struct sk_buff *skb)
265     {
266             u16 key = teredo_phash (v6addr);
267             skb_queue_tail(&(teredo_peer[key].pktq), (struct sk_buff *) skb);
268             return;
269     }
270
271     static inline struct sk_buff*
272     teredo_dequeue (const struct in6_addr *v6addr)
273     {
274             u16 key = teredo_phash (v6addr);
275             struct sk_buff* skb = skb_dequeue(&(teredo_peer[key].pktq));
276             return skb;
277     }
278
279     static inline int
280     teredo_isqempty (const struct in6_addr *v6addr)
281     {
282             u16 key = teredo_phash (v6addr);
283             return skb_queue_empty(&(teredo_peer[key].pktq));
284     }
285
286     #endif
```

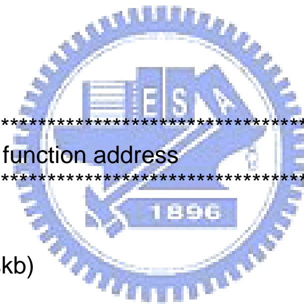
C.3 teredo.c

```
1       /*
2        *   Teredo tunnel device - Tunneling IPv6 over UDP through NATs
3        *   Linux INET6 implementation
4        *
5        *
6        *       $Id: teredo.c,v 1.33 2005/03/10 12:49:56 smhuang Exp $
```

```
7         *
8         *   This program is free software; you can redistribute it and/or
9         *   modify it under the terms of the GNU General Public License
10        *   as published by the Free Software Foundation; either version
11        *   2 of the License, or (at your option) any later version.
12        *
13        *   Reference: net/ipv6/sit.c
14        *
15        *   Version 0.0 - 10/24/03 - Initial Version.
16        *               0.1 - 12/24/03 - Add Teredo Bubble support.
17        *               0.2 - 01/12/04 - Bug Fix (tunnel packet format in udpip6_tunnel_xmit).
18        *               0.3 - 03/16/04 - Add kernel thread for Bubble retransmission.
19        *               0.4 - 01/15/05 - Remove kernel thread feature.
20        *               0.5 - 03/11/05 - Add packet check feature.
21        */
22
23
24
25       #include <linux/module.h>
26       #include <asm/uaccess.h>
27       #include <linux/netfilter_ipv4.h>
28
29       #include <net/ipv6.h>
30       #include <net/protocol.h>
31       #include <net/ip6_route.h>
32       #include <net/ndisc.h>
33       #include <net/addrconf.h>
34       #include <net/ip.h>
35       #include <net/udp.h>
36       #include <net/icmp.h>
37       #include <net/ipip.h>
38       #include <net/inet_ecn.h>
39
40       #include "teredo.h"
41       #include "bubble.h"
42
43
44       /*
45        *****************************************************************
46        * Decapsulates the IPv4 and UDP headers of skb to IPv6                 *
47        * 1) check validity                                                     *
48        * 2) add an entry in teredo peer list if receives a bubble        *
49        * 3) remove the IPv4 and UDP header                                          *
50        * 4) call netif_rx() for delivery to a normal IPv6 host            *
51        *****************************************************************
52        */
53       int
54       udpip6_rcv(struct sk_buff *skb)
55       {
56             struct iphdr *iph;
57             struct udphdr *uh;
58             struct ipv6hdr *iph6;
59             struct ip_tunnel *tunnel;
60
61       DEBUG_TRACE("TEREDO:udpip6_rcv\n");
62             if (!pskb_may_pull(skb, sizeof(struct ipv6hdr)))
63             {
64                   goto out;
65             }
66
67             iph = skb->nh.iph;
68             read_lock(&udpip6_lock);
69             if(skb->h.uh!=NULL)
70             {
71                   uh = skb->h.uh;
72
```

```
73              /* check source and dest, dest must be 3544 */
74              if (htons(uh->dest) != TEREDO_PORT)
75              {
76                  goto fout;
77              }
78          }
79          else
80          {
81              goto fout;
82          }
83
84          if (teredo_tunnel != NULL)
85          {
86              tunnel = teredo_tunnel;
87              skb->data = skb_pull(skb, sizeof(struct udphdr));
88              skb->mac.raw = skb->nh.raw;
89              skb->nh.raw = skb->data;
90
91              iph6 = skb->nh.ipv6h;
92              /* check formal teredo src IPv6 addr --ijk--*/
93              if (!teredo_isaddr(&iph6->saddr))
94              {
95                  goto fout;
96              }
97
98              /* check mapped transport addr matches field in src teredo addr --ijk--*/
99              if (ntohl(iph->saddr) != iph6->saddr.in6_u.u6_addr32[3] ||
100                 ntohs(uh->source) != iph6->saddr.in6_u.u6_addr16[6] )
101             {
102                 kfree_skb(skb);
103                 read_unlock(&udpip6_lock);
104                 return 0;
105             }
106
107             if (teredo_ispeer(&iph6->saddr))
108             {
109                 teredo_settime_rx(&iph6->saddr);
110
111                 if (!teredo_istrust(&iph6->saddr))
112                 {
113                     teredo_trust(&iph6->saddr);
114                     teredo_add_mapped(&iph6->saddr, iph->saddr, uh->source);
115                 }
116
117                 /* discard if is bubble */
118                 if (iph6->nexthdr == IPPROTO_NONE)
119                 {
120                     kfree_skb(skb);
121                     while(!teredo_isqempty(&iph6->saddr))
122                     {
123                         skb = teredo_dequeue(&iph6->saddr);
124                         dev_queue_xmit(skb);
125                     }
126                     read_unlock(&udpip6_lock);
127                     return 0;
128                 }
129             }
130             else
131             {
132                 /* discard if not peer */
133                 goto fout;
134             }
135
136             memset(&(IPCB(skb)->opt), 0, sizeof(struct ip_options));
137             skb->protocol = htons(ETH_P_IPV6);
138             skb->pkt_type = PACKET_HOST;
```

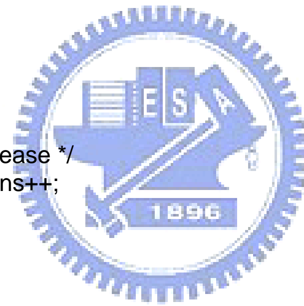
```
139                 tunnel->stat.rx_packets++;
140                 tunnel->stat.rx_bytes += skb->len;
141                 skb->dev = tunnel->dev;
142                 dst_release(skb->dst);
143                 skb->dst = NULL;
144     #ifdef CONFIG_NETFILTER
145                 nf_conntrack_put(skb->nfct);
146                 skb->nfct = NULL;
147     #ifdef CONFIG_NETFILTER_DEBUG
148                 skb->nf_debug = 0;
149     #endif
150     #endif
151                 if (INET_ECN_is_ce(iph->tos) && INET_ECN_is_not_ce(ip6_get_dsfield(skb->nh.ipv6h)))
152                 {
153                     IP6_ECN_set_ce(skb->nh.ipv6h);
154                 }
155                 netif_rx(skb);
156                 read_unlock(&udpip6_lock);
157                 return 0;
158         }
159
160         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PROT_UNREACH, 0);
161     fout:
162         kfree_skb(skb);
163         read_unlock(&udpip6_lock);
164     out:
165         return 0;
166     }
167
168
```




```
169     /*
170      ******************************************************************
171      * for NF_HOOK to take the function address                        *
172      ******************************************************************
173      */
174     static inline int
175     do_ip_send(struct sk_buff *skb)
176     {
177     DEBUG_TRACE("TEREDO:do_ip_send\n");
178         return ip_send(skb);
179     }
180
181
182     /*
183      ******************************************************************
184      * Encapsulates the skb in IPv4 UDP                                 *
185      * 1) check validity                                                *
186      * 2) send teredo bubble according to the draft if necessary       *
187      * 3) push down and install the IPv4 and UDP header                 *
188      * 4) call IPTUNNEL_XMIT() for delivery to a teredo client         *
189      ******************************************************************
190      */
191     static int
192     udpip6_tunnel_xmit(struct sk_buff *skb, struct net_device *dev)
193     {
194         struct ip_tunnel *tunnel = (struct ip_tunnel*)dev->priv;
195         struct net_device_stats *stats = &tunnel->stat;
196         struct iphdr    *tiph = &tunnel->parms.iph;
197         struct ipv6hdr *iph6 = skb->nh.ipv6h;
198         u8        tos = tunnel->parms.iph.tos;
199         struct rtable *rt;                  /* Route to the other host */
200         struct net_device *tdev;            /* Device to other host */
201         struct iphdr    *iph;               /* Our new IP header */
202         int       max_headroom;             /* The extra header space needed */
203         u32       dst = tiph->daddr;
204         int       mtu;
```

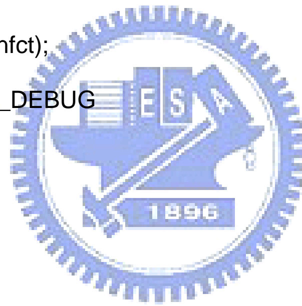
```
205
206             struct udphdr *uh;
207             int         teredo_send_bubble = 0;    //flag for bubble
208             struct sk_buff *bubble = NULL;
209
210         DEBUG_TRACE("TEREDO:udpip6_tunnel_xmit\n");
211             if (tunnel->recursion++)
212             {
213                 /* xmit is processing another pkt */
214                 tunnel->stat.collisions++;
215                 goto tx_error;
216             }
217
218             if (skb->protocol != htons(ETH_P_IPV6))
219             {
220                 goto tx_error;
221             }
222
223
224             /* check valid dest teredo IPv6 addr --ijk--*/
225             if (!teredo_isaddr(&iph6->daddr))
226             {
227                 dev_kfree_skb(skb);
228                 tunnel->recursion--;
229                 return 0;
230             }
231
232             // for NAT traversal    //--ijk-- SMTEST
233             if (teredo_iscone(&iph6->daddr))
234             {
235                 // case 2) -- cone bit is set
236                 if (!teredo_ispeer(&iph6->daddr))
237                 {
238                     teredo_new_cone_peer(&iph6->daddr);
239                 }
240                 dst = try_teredo(&iph6->daddr);
241                 teredo_settime_tx(&iph6->daddr);
242             }
243             else
244             {
245                 if (teredo_ispeer(&iph6->daddr))
246                 {
247                     if (teredo_istrust(&iph6->daddr))
248                     {
249                         // case 1) trust
250                         teredo_settime_tx(&iph6->daddr);
251                     }
252                     else
253                     {
254                         // in the process of sending bubble??
255                         DEBUG_TRACE("TEREDO:case bubble\n");
256                         goto tx_error;
257                     }
258                 }
259                 else
260                 {
261                     // case 3) -- transmit bubble
262                     teredo_new_peer(&iph6->daddr);
263                     teredo_send_bubble = 1;
264                     bubble = skb_copy(skb, GFP_ATOMIC);
265                     teredo_enqueue(&iph6->daddr, skb);
266
267                     teredo_build_bubble(bubble);
268                     skb = bubble;
269                     teredo_inc_bubble(&iph6->daddr);
270                 }
```

```
271         }
272 
273         /* fill in dest. addr. in teredo style*/
274         if (teredo_send_bubble)
275         {
276             dst = teredo_server_addr(&iph6->daddr);
277         }
278         else
279         {
280             dst = try_teredo(&iph6->daddr);
281         }
282 
283         if (0 == dst)
284         {
285             /* fail to restore the dest. addr. from teredo addr. */
286             goto tx_error;
287         }
288         /* choose an appropriate routing table */
289         if (ip_route_output(&rt, dst, tiph->saddr, RT_TOS(tos), tunnel->parms.link))
290         {
291             tunnel->stat.tx_carrier_errors++;
292             goto tx_error_icmp;
293         }
294         if (rt->rt_type != RTN_UNICAST)
295         {
296             tunnel->stat.tx_carrier_errors++;
297             goto tx_error_icmp;
298         }
299         tdev = rt->u.dst.dev;
300 
301         if (tdev == dev)
302         {
303             ip_rt_put(rt);    /* release */
304             tunnel->stat.collisions++;
305             goto tx_error;
306         }
307 
308         if (tiph->frag_off)
309         {
310             mtu = rt->u.dst.pmtu - sizeof(struct iphdr) - sizeof(struct udphdr);
311         }
312         else
313         {
314             mtu = skb->dst ? skb->dst->pmtu : dev->mtu;
315         }
316 
317         if (mtu < 68)
318         {
319             tunnel->stat.collisions++;
320             ip_rt_put(rt);
321             goto tx_error;
322         }
323         if (mtu < IPV6_MIN_MTU)
324         {
325             mtu = IPV6_MIN_MTU;
326         }
327         if (skb->dst && mtu < skb->dst->pmtu)
328         {
329             struct rt6_info *rt6 = (struct rt6_info*)skb->dst;
330             if (mtu < rt6->u.dst.pmtu)
331             {
332                 if (tunnel->parms.iph.daddr || rt6->rt6i_dst.plen == 128)
333                 {
334                     rt6->rt6i_flags |= RTF_MODIFIED;
335                     rt6->u.dst.pmtu = mtu;
336                 }
```

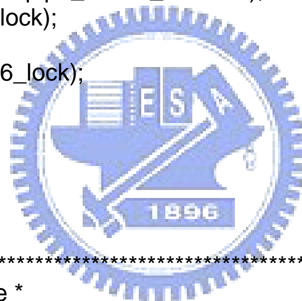


```
337                 }
338         }
339         if (skb->len > mtu)
340         {
341                 icmpv6_send(skb, ICMPV6_PKT_TOOBIG, 0, mtu, dev);
342                 ip_rt_put(rt);
343                 goto tx_error;
344         }
345
346         if (tunnel->err_count > 0)
347         {
348                 if (jiffies - tunnel->err_time < IPTUNNEL_ERR_TIMEO)
349                 {
350                         tunnel->err_count--;
351                         dst_link_failure(skb);
352                 }
353                 else
354                 {
355                         tunnel->err_count = 0;
356                 }
357         }
358         skb->h.raw = skb->nh.raw;
359
360         /* see if we can stuff it in the buffer*/
361         max_headroom = (((tdev->hard_header_len+15)&~15)+sizeof(struct iphdr)+sizeof(struct udphdr));
362
363         if (skb_headroom(skb) < max_headroom || skb_cloned(skb) || skb_shared(skb)) {
364                 struct sk_buff *new_skb = skb_realloc_headroom(skb, max_headroom);
365                 if (!new_skb)
366                 {
367                         ip_rt_put(rt);
368                         stats->tx_dropped++;
369                         dev_kfree_skb(skb);
370                         tunnel->recursion--;
371                         return 0;
372                 }
373                 if (skb->sk)
374                 {
375                         skb_set_owner_w(new_skb, skb->sk);
376                 }
377                 dev_kfree_skb(skb);
378                 skb = new_skb;
379                 iph6 = skb->nh.ipv6h;
380         }
381
382         skb->ip_summed = CHECKSUM_UNNECESSARY;
383         skb->h.raw = skb_push(skb, sizeof(struct udphdr));
384         skb->nh.raw = skb_push(skb, sizeof(struct iphdr));
385         memset(&(IPCB(skb)->opt), 0, sizeof(IPCB(skb)->opt));
386         dst_release(skb->dst);
387         skb->dst = &rt->u.dst;
388
389
390         /* push down and install the UDP header */
391         uh = skb->h.uh;
392         uh->source = htons(TEREDO_PORT);
393         if (teredo_iscone(&iph6->daddr)) //--ijk-- SMTEST
394         {
395                 uh->dest = teredo_port(&iph6->daddr);
396         }
397         else if (teredo_send_bubble)
398         {
399                 uh->dest = htons(TEREDO_PORT);//to server
400         }
401         else
402         {
```

```
403                     uh->dest = teredo_peer_port(&iph6->daddr);//use mapped port in peer list
404               }
405               uh->len = htons(skb->len-sizeof(struct iphdr));
406               uh->check = 0;
407
408               /* push down and install the IPv4 header */
409               iph                    =     skb->nh.iph;
410               iph->version           =     4;
411               iph->ihl               =     sizeof(struct iphdr)>>2;
412               if (mtu > IPV6_MIN_MTU)
413               {
414                     iph->frag_off = htons(IP_DF);
415               }
416               else
417               {
418                     iph->frag_off = 0;
419               }
420               iph->protocol          =     IPPROTO_UDP;
421               iph->tos               =     INET_ECN_encapsulate(tos, ip6_get_dsfield(iph6));
422               iph->daddr             =     rt->rt_dst; //--ijk-- mapped may not useful!!!
423               iph->saddr             =     rt->rt_src;
424
425               if ((iph->ttl = tiph->ttl) == 0)
426               {
427                     iph->ttl      = iph6->hop_limit;
428               }
429
430         #ifdef CONFIG_NETFILTER
431               nf_conntrack_put(skb->nfct);
432               skb->nfct = NULL;
433         #ifdef CONFIG_NETFILTER_DEBUG
434               skb->nf_debug = 0;
435         #endif
436         #endif
437
438               IPTUNNEL_XMIT();
439               tunnel->recursion--;
440
441               return 0;
442
443         tx_error_icmp:
444               dst_link_failure(skb);
445         tx_error:
446               stats->tx_errors++;
447               dev_kfree_skb(skb);
448               tunnel->recursion--;
449               return 0;
450         }
451
452
453         static struct
454         net_device_stats *udpip6_tunnel_get_stats(struct net_device *dev)
455         {
456         DEBUG_TRACE("TEREDO:udpip6_tunnel_get_stats\n");
457               return &(((struct ip_tunnel*)dev->priv)->stat);
458         }
459
460
461         static int
462         udpip6_tunnel_change_mtu(struct net_device *dev, int new_mtu)
463         {
464         DEBUG_TRACE("TEREDO:udpip6_tunnel_change_mtu\n");
465               if (new_mtu < IPV6_MIN_MTU || new_mtu > TEREDO_MTU)
466               {
467                     return -EINVAL;
468               }
```



```
469         dev->mtu = new_mtu;
470         return 0;
471     }
472
473
474     static int
475     udpip6_unuse_module(struct net_device *dev)
476     {
477     DEBUG_TRACE("TEREDO:udpip6_tunnel_close\n");
478         MOD_DEC_USE_COUNT;
479         return 0;
480     }
481
482
483     static int
484     udpip6_use_module(struct net_device *dev)
485     {
486     DEBUG_TRACE("TEREDO:udpip6_tunnel_open\n");
487         MOD_INC_USE_COUNT;
488         return 0;
489     }
490
491
492     static void
493     udpip6_tunnel_uninit(struct net_device *dev)
494     {
495     DEBUG_TRACE("TEREDO:udpip6_tunnel_uninit\n");
496         write_lock_bh(&udpip6_lock);
497         teredo_tunnel = NULL;
498         write_unlock_bh(&udpip6_lock);
499         dev_put(dev);
500         return;
501     }
502
503
504     /*
505      **************************************************************************
506      * Initial Teredo tunnel device *
507      **************************************************************************
508      */
509     int
510     udpip6_tunnel_init(struct net_device *dev)
511     {
512         struct iphdr *iph;
513     DEBUG_TRACE("TEREDO:udpip6_tunnel_init\n");
514         struct ip_tunnel *t    = (struct ip_tunnel*)dev->priv;
515
516         dev->destructor          = 0;
517         dev->uninit              = udpip6_tunnel_uninit;
518         dev->hard_start_xmit = udpip6_tunnel_xmit;
519         dev->get_stats           = udpip6_tunnel_get_stats;
520         dev->do_ioctl            = 0;
521         dev->change_mtu          = udpip6_tunnel_change_mtu;
522
523         dev->type                = ARPHRD_SIT;
524         dev->hard_header_len = LL_MAX_HEADER + sizeof(struct iphdr) + sizeof(struct udphdr);
525         dev->mtu                 = TEREDO_MTU;
526         dev->flags               = IFF_NOARP;
527         dev->iflink              = 0;
528         dev->addr_len            = 4;
529         memcpy(dev->dev_addr, &t->parms.iph.saddr, 4);
530         memcpy(dev->broadcast, &t->parms.iph.daddr, 4);
531         dev->open = udpip6_use_module;
532         dev->stop = udpip6_unuse_module;
533
534         iph                  = &udpip6_tunnel.parms.iph;
```



```
535         iph->version    = 4;
536         iph->protocol = IPPROTO_UDP;
537         iph->ihl          = 5;
538         iph->ttl          = 64;
539
540         dev_hold(dev);
541         teredo_tunnel = &udpip6_tunnel;
542         return 0;
543     }
544
545
546     /*
547      ****************************************************************
548      * Unregister Teredo tunnel device                                                *
549      ****************************************************************
550      */
551     void
552     cleanup_module(void)
553     {
554     DEBUG_TRACE("TEREDO:cleanup_module\n");
555         inet_del_protocol(&teredo_protocol);
556         unregister_netdev(&udpip6_tunnel_dev);
557     }
558
559
560     /*
561      ****************************************************************
562      * Regester and initializes Teredo tunnel device                            *
563      ****************************************************************
564      */
565     int
566     init_module(void)
567     {
568     DEBUG_TRACE("TEREDO:init_module\n");
569         printk(KERN_INFO "IPv6 over UDP tunneling driver\n");
570
571         udpip6_tunnel_dev.priv = (void*)&udpip6_tunnel;
572         strcpy(udpip6_tunnel_dev.name, udpip6_tunnel.parms.name);
573         register_netdev(&udpip6_tunnel_dev);
574         inet_add_protocol(&teredo_protocol);
575
576         return 0;
577     }
578
579
580     MODULE_DESCRIPTION("Teredo IPv6 Tunneling module");
581     MODULE_LICENSE("GPL");
```