第 7 部

次世代インターネットプロトコル

第 1 章

v6 分科会

この章では、IPv6 と IPsec に関する研究に取り組んでいる v6 分科会が 1998 年度に活動した内容について報告する。

1.1 概要

1998 年 4 月から、IPv6 と IPsec のスタックを集中的に開発する KAME プロジェクトが始まった。KAME プロジェクトには、v6 分科会で活発に活動していた 7 組織が参加している。初期段階では、v6 分科会の参照コードであった Hydrangea に、各組織で実装された機能をマージした。マージの作業が終ってからは、マージされたコードの改良や新たな機能の追加に取り組んでいる。

KAME プロジェクトは、毎週 snap バージョン、2 ヵ月に 1 度 stable バージョンをリリースしている。現在このコードが v6 分科会の参照コードとなっている。v6 分科会では、今年度はトランスレータの実装や運用に力を入れた。トランスレータや個々に開発されたアプリケーションの多くは、KAME のコードに取り込まれ、ports として配布されている。

このような活動に伴い、KAME のコードの論文やトランスレータの Internet-Draft が出版された。これらの文献を取りまとめる形でこの章は以下のように構成される。

- TCP とエニーキャストに関する Internet-Draft
- トランスレータの分類に関する Internet-Draft
- Bump In The Stack というトランスレータに関する Internet-Draft
- SOCKS に基づいたトランスレータに関する Internet-Draft
- KAME コードの実装方法に関する INET'99 の論文
- 1 年間の活動状況

1.2 Disconnecting TCP connection toward IPv6 anycast address

1.2.1 Abstract

IPv6 specification implicitly disallows TCP connection toward IPv6 anycast address. However, if such a connection request happens by mistake, currently there is no way to report the incident to the originator of the TCP connection. The document tries to define a way to disconnect TCP connections made toward IPv6 anycast addresses.

1.2.2 Problem

IPv6 specification [42] defines "anycast address". Anycast addresses have the following capabilities and restrictions:

- Anycast address is not distinguishable from non-anycast, unicast addresses.
- Anycast address can be assigned to multiple interfaces of multiple nodes.
- Anycast address MUST NOT be assigned to an IPv6 host. It can be assigned to an IPv6 router only.
- Anycast address MUST NOT be used in source address field in IPv6 header.

Since anycast address MUST NOT be used as IPv6 source address, TCP connection using anycast address MUST NOT be established. However, since anycast address is indistinguishable from (non-anycast) unicast addresses, a TCP connection is sometimes requested toward an anycast address, and in the TCP scheme there is no way to report the error.

Consider the following scenario:

- We have Host A and router B on the network. The topology between host A and router B has no effect to the following scenario, so we assume they share the same network medium.
- An IPv6 anycast address, `3ffe:0501::1`, is assigned to an interface of router B.

Since an anycast address is indistinguishable from non-anycast addresses, host A may transmit TCP connection request toward IPv6 address `3ffe:0501::1`, port X.

The standard behavior for TCP stack on router B is to transmit toward host A a TCP packet, with RST bit set to disconnect the TCP connection. However, in this situation

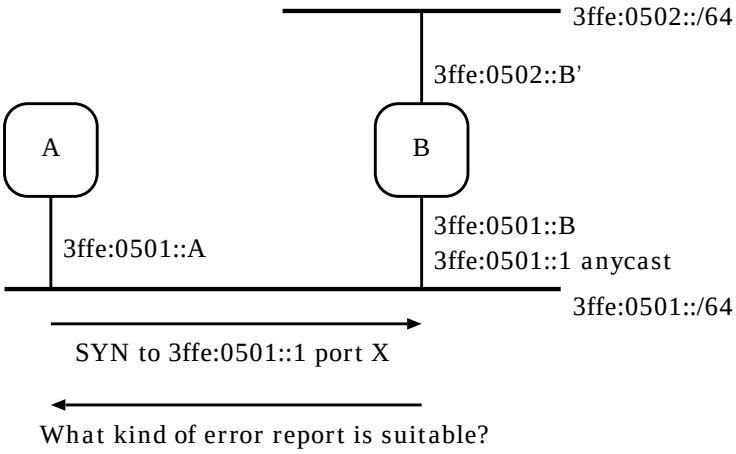


図 1.1: TCP connection request toward anycast address.

router B cannot transmit TCP RST packet toward host A, since router B needs to put its anycast address, `3ffe:0501::1`, into source address field in IPv6 header to do this.

A non-active way of dealing with this problem is, silently ignore the TCP connection request from host A to router B, and wait for host A to timeout. This is not a desired behavior.

1.2.3 Solution

If a router gets TCP connection request (SYN packet) toward one of its anycast addresses, the router SHOULD transmit an ICMPv6 [64] error packet, with type field 1 (destination unreachable), code field 3 (address unreachable). Source address field of IPv6 header will be filled by an IPv6 (non-anycast) unicast address assigned to the router.

In the example illustrated in Problem section, router B SHOULD transmit an ICMPv6 error packet toward host A. The source address field of IPv6 header for the ICMPv6 packet will be `3ffe:0501::B`.

Implementation note: If the originating host of the TCP connection is a BSD-based implementation, it will count the number of ICMPv6 error packets in `tcp_notify()`. If the number of ICMPv6 error packets exceeds the limit, the originating host drops the TCP connection. In this case, TCP connection can be disconnected much quicker.

The proposed method MUST be removed when one of the following events happens in the future:

- Restriction imposed on IPv6 anycast address is loosened, so that anycast address can be placed into source address field of the IPv6 header, or

- TCP-over-IPv6 is modified to provide some way to accept a connection toward IPv6 anycast address.

1.2.4 Alternative solutions

We can try to define some DNS resource record to denote if an IPv6 unicast address is anycast address, or non-anycast address. However, it looks to be very hard to deploy. Also, we must have a way to disconnect TCP connection request toward anycast address, anyway.

1.2.5 Security considerations

The document only clarifies the use of TCP-over-IPv6 and ICMPv6, and the author believes that the draft raises no new security problem.

Malicious intermediate router can transmit forged ICMPv6 error report packet, to prevent two hosts from establishing TCP connection. This is known problem in ICMPv6, and ICMPv6 specification [64] provides a detailed discussions on this problem (see section 5 in RFC1885 [37]).

ICMPv6 address unreachable packet provides a way to report an error in per-address, not per-TCP-port, basis. Therefore, some implementation may lose all connections toward the reported address, not the specific connection that caused the ICMPv6 error report. This is not a problem for this draft, because no TCP connection toward IPv6 anycast address is allowed at this moment.

1.3 Categorizing Translators between IPv4 and IPv6

1.3.1 Abstract

This memo categorizes translators between IPv4 and IPv6. The two components, address interpretation and address mapping, are discussed. This draft is based on a paper appeared in the proceedings of INET98[136]. The intention of this memo is circulation of such knowledge.

1.3.2 Introduction

In the early stage of the migration from IPv4[107] to IPv6[42], it is expected that IPv6 islands will be connected to the IPv4 ocean. On the other hand, in the late stage of the migration, IPv4 islands will be connected to the IPv6 ocean. IPv4 hosts need to be

connected to the Internet after the IPv4 address space is exhausted. So, it is necessary to develop translators to enable direct communication between IPv4 hosts and IPv6 hosts.

This memo assumes the following for the practical migration scenario from IPv4 to IPv6:

1. We cannot modify IPv4 hosts, but we can implement IPv6 hosts as we like.
2. A small space of IPv4 address is also assigned to an IPv6 island according to the current severe address assignment policy.
3. An IPv4 island can also obtain a large space of IPv6 address.

A typical translator consists of two components: interpretation between IPv4 packets and IPv6 packets described in Section 1.3.3 and address mapping between IPv4 and IPv6 explained in Section 1.3.4.

1.3.3 Interpretation of IPv4 and IPv6

For interpretation of IPv4 and IPv6, three technologies are available: header conversion, transport relay, and application proxy.

Header Conversion

Header conversion refers to converting IPv6 packet headers to IPv4 packet headers, or vice versa, and adjusting (or re-calculating) checksums if necessary. This is IP level translation. Examples are SIIT[98] and main part of NATPT[124]. Note that NAT[47] is IPv4-to-IPv4 header converter.

Header conversion is fast enough, but it has disadvantages in common with NAT. A good example is difficulty in the translation of network layer addresses embedded in application layer protocols, which are typically found in FTP and FTP Extensions[14].

Also, header conversion has problems which are not found in NAT: a large IPv4 packet is fragmented to IPv6 packets because the header length of IPv6 is typically 20 bytes larger than that of IPv4, and the semantics of ICMP[106] and that of ICMPv6[39] are not inter-changeable.

Transport relay

Transport relay refers to relaying a TCP, UDP/IPv4 session and a TCP, UDP/IPv6 session in the middle. This is transport level translation.

For example, a typical TCP relay server works as follows: when a TCP request reaches a relay server, the network layer tosses it up to the TCP layer even if the destination is not the server's address. The server accepts this TCP packet and establishes a TCP connection from the source host. One more TCP connection is also made from the server to the real

destination. Then the server reads data from one of the two connections and writes the data to the other.

SOCKS[77] is an another example. A SOCKS based translator requires client hosts to be "SOCKS-ready" by installing SOCKS libraries, etc.

Transport relay does not have problems like fragmentation or ICMP

conversion, since each session is closed in IPv4 and IPv6, respectively, but it does have problems like the translation of network layer addresses embedded in application layer protocols.

Application Proxy

An application proxy for a transaction service is used to hide site information and improve service performance with a cache mechanism. An application proxy can be a translator between IPv4 and IPv6 if it supports both protocols. This is application level translation.

Since each service is closed in IPv4 and IPv6, respectively, there are no disadvantages found in header conversion, but servers for each service must be bilingual.

1.3.4 Address Mapping

Address mapping refers to the allocation of an IPv6 destination address for a given IPv4 destination address, and vice versa. It also includes the allocation of an IPv6 source address for a given IPv4 source address, and vice verca. If interpretation is performed at the Internet protocol level or transport level, address mapping is an essential issue.

If an FQDN(Fully Qualified Domain Name) is used to specify a target host, address mapping is not necessary. So, the application proxy is free of this problem. SOCKS version 5 is a kind of TCP relay, but it is also free of this because it can make use of FQDN.

In the case that address mapping is dynamic, it must be implemented in interaction with DNS. However, if it is static and proliferation of mapped addresses is limited to a small region(i.e. Translator A, described later), it can be implemented by extending resolver libraries on local hosts. Of course, DNS can also map addresses statically.

An example of library extensions for static mapping: an application tries resolving `AAAA` records against a host name. The resolver library requests DNS servers `A` or `AAAA` records of the name. If only `A` records are returned, the library converts them to `AAAA` records embedding them into the pre-configured prefix. ([124] also discusses this mechanism.)

An example of DNS extensions for dynamic mapping: if a DNS server receives a request to return `A` records for a host name, but only an `AAAA` record is resolved, the server picks up an IPv4 address from its address pool then returns it as `A` record.

There are two criteria for addresses to be assigned: (1) the assigned addresses must

be reachable between the triggered host and translator, and (2) if addresses are assigned dynamically by DNS, it must be ensured that the DNS cache doesn't cause problems for further communications.

If transport relay is used for interpretation, address mapping is necessary only for destination addresses since source address mapping is closed in the relay server. In other words, the protocol association of the first transport session is mapped to a local port number on the relay server.

For header conversion, source address mapping is not essential, either. A protocol association can be represented by a local port of the conversion router or by an address out of the pool or by both.

Translator Categories

To discuss address mapping, this memo categorizes IPv4/IPv6 translator into four types illustrated by the following pictures:

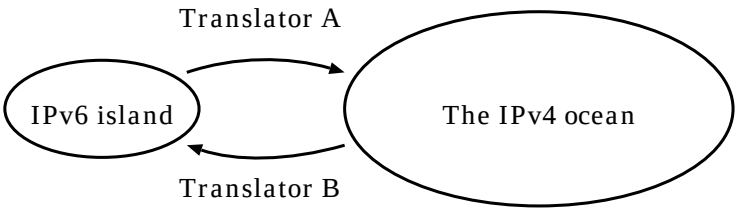


図 1.2: In the early stage

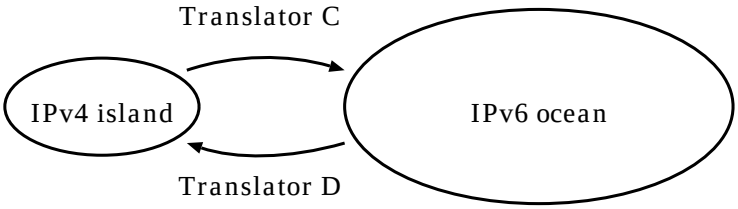


図 1.3: In the late stage

Translator A It is used in the early stage of transition to establish a connection from an IPv6 host in an IPv6 island to an IPv4 host in the IPv4 ocean.

Translator B It is used in the early stage of transition to establish a connection from an IPv4 host in the IPv4 ocean to an IPv6 host in an IPv6 island.

Translator C It is used in the late stage of transition to establish a connection from an IPv4 host in an IPv4 island to an IPv6 host in the IPv6 ocean.

Translator D It is used in the late stage of transition to establish a connection from an IPv6 host in the IPv6 ocean to an IPv4 host in an IPv4 island.

Observations on Address Mapping for Each Translator

Here are observations on address mapping for each translator:

Translator A Destination address mapping: global IPv4 to global IPv6 Static or dynamic: static Address pool: a part of assigned global IPv6 addresses to the IPv6 site DNS cache problem: not encountered Implementation: straightforward Note: IPv4 addresses can be embedded to pre-configured IPv6 prefix.

Translator B Destination address mapping: global IPv6 to global IPv4 Static or dynamic: dynamic Address pool: assigned global IPv4 addresses to the IPv6 site DNS cache problem: potentially proliferated into the IPv4 ocean Implementation: nearly impossible Note: it is recommended to use static address mapping for several IPv6 hosts(servers) in the IPv6 site to provide their services to the IPv4 ocean(e.g. dual-stack servers without translators).

Translator C Destination address mapping: global IPv6 to private IPv4 Static or dynamic: dynamic Address pool: a part of private IPv4 addresses DNS cache problem: closed to the IPv4 site Implementation: possible Note: mapped addresses should be reserved as long as possible for UDP applications which can't tell the end of communications and for applications which cache DNS entries.

Translator D Destination address mapping: global IPv4 to global IPv6 Static or dynamic: static Address pool: assigned global IPv6 addresses to the site DNS cache problem: not encountered Implementation: straightforward Note: IPv4 addresses can be embedded to pre-configured IPv6 prefix.

1.4 Dual Stack Hosts using the "Bump-in-the-Stack" Technique

1.4.1 Abstract

Especially in the early stage of the migration from IPv4 to IPv6, it is hard to prepare IPv6 applications completely. This memo proposes a mechanism of dual stack hosts using

the technique called “Bump-in-the-Stack” in the IP security area. The mechanism enables the hosts to communicate with other IPv6 hosts using IPv4 legacy applications.

1.4.2 Introduction

RFC1933 [58] proposed mechanisms to migrate from IPv4 [107] to IPv6 [42], including dual stack and tunneling, for the early stage. Accordingly, hosts and routers are developed for the IPv6 migration. But there are few applications for IPv6 compared to IPv4, where a huge number of applications are available. In order to advance the migration to IPv6 smoothly, it is highly desirable to increase the availability of IPv6 applications to the same level as IPv4. But unfortunately this is expected to take a very long time.

This memo proposes a mechanism of dual stack hosts using the technique called “Bump-in-the-Stack” in the IP security area. The technique inserts modules into the hosts which snoop data that flows between a TCP/IPv4 module and network card driver modules, and translate IPv4 into IPv6 and vice versa. It enables the hosts to communicate with other IPv6 hosts using IPv4 legacy applications; thus making it seem as if the hosts have applications for both IPv4 and IPv6.

This document uses the words defined in [107], [42], and [58].

1.4.3 Components

Dual stack hosts defined in RFC1933 [58] need applications, TCP/IP modules and addresses for both IPv4 and IPv6. The proposed hosts in this memo have 3 modules instead of IPv6 applications, and communicate with other IPv6 hosts using IPv4 applications. The 3 modules are a translator, an extension name resolver and an address mapper.

Figure 1.4 illustrates a host which has the 3 modules described above installed.

Translator

The translator translates IPv4 into IPv6 and vice versa using the IP conversion mechanism defined in [98].

When receiving IPv4 packets from IPv4 applications, the translator converts IPv4 packet headers into IPv6 packet headers, then fragments the IPv6 packets (because header length of IPv6 is typically 20 bytes larger than that of IPv4), and sends them to IPv6 networks. When receiving IPv6 packets from the IPv6 networks, it works symmetrically to the previous case, except that there is no need to fragment the packets.

Extension Name Resolver

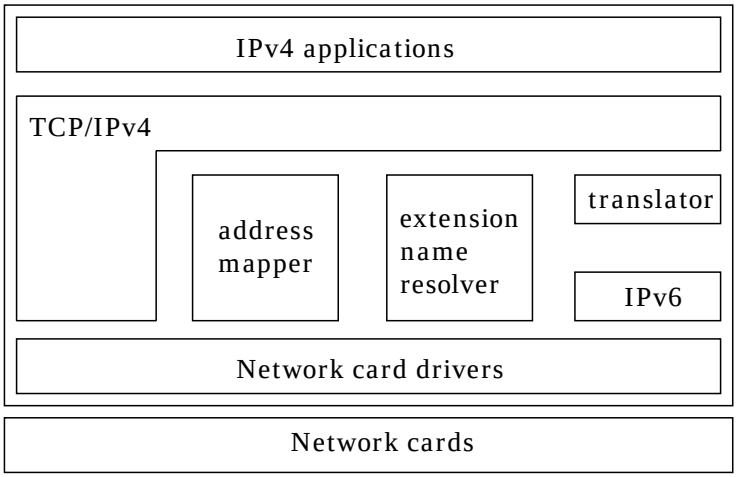


図 1.4: The proposed dual stack host

The extension name resolver returns a "proper" answer in response to the IPv4 application's request.

The application typically sends a query to its name server to resolve `A` records for the target host name. The translator snoops the query, then creates another query to resolve both `A` and

`AAAA` records for the host name, and sends the query to the server. If the `A` record is resolved, it returns the `A` record to the application. In this case, there is no need for translation by the translator above. If only the `AAAA` record is available, it requests the mapper to assign an IPv4 address corresponding to the IPv6 address. Then it creates the `A` record for the assigned IPv4 address and returns the `A` record to the application.

Address mapper

The address mapper maintains an IPv4 address spool. The spool, for example, consists of private addresses [110]. Also, it maintains pairs consisting of an IPv4 address and an IPv6 address in a table.

When the resolver or the translator requests the mapper to assign an IPv4 address for an IPv6 address, it selects and returns an IPv4 address out of the spool, and then registers a new entry into the table dynamically. The registration occurs in the following 2 cases:

1. When the resolver gets only an `AAAA` record for the target host name and there is not a mapping entry for the IPv6 address.

2. When the translator receives an IPv6 packet and there is not a mapping entry for the IPv6 source address.

NOTE: There is one exception to above. When initializing the table, it registers a pair of its own IPv4 address and IPv6 address into the table statically.

1.4.4 Action Examples

This section describes action of the proposed dual stack host called "dual stack," which communicates with an IPv6 host called "host6" using an IPv4 application.

Originator behavior

This subsection describes the originator behavior of "dual stack." The communication is triggered by "dual stack."

The application sends a query to its name server to resolve `A` records for "host6."

The resolver snoops the query, and then creates another query to resolve both `A` and `AAAA` records for the host name and sends it to the server. In the case, only the `AAAA` record is resolved, so the resolver requests the mapper to assign an IPv4 address corresponding to the IPv6 address.

NOTE: In the case of communication with an IPv4 host, the `A` record is resolved. The resolver then returns it to the application, and there is no need for translation as follows.

The mapper selects an IPv4 address out of the spool and returns it to the resolver.

The resolver creates the `A` record for the assigned IPv4 address and returns it to the application.

NOTE: See subsection 1.4.5 about influence on other hosts caused by the assigned IPv4 address.

The application sends an IPv4 packet to "host6."

The IPv4 packet reaches the translator. The translator tries translating the IPv4 packet into an IPv6 packet but does not know how to translate the IPv4 destination address and the IPv4 source address. So the translator requests the mapper to provide mapping entries for them.

The mapper checks its mapping table and finds entries for each of them, and then returns the IPv6 destination address and the IPv6 source address to the translator.

NOTE: The mapper will register its own IPv4 address and IPv6 address into the table beforehand.

The translator translates the IPv4 packet into an IPv6 packet then fragments the IPv6 packet if necessary and sends it to an IPv6 network.

The IPv6 packet reaches "host6." Then "host6" sends a new IPv6 packet to "dual stack."

The IPv6 packet reaches the translator of "dual stack." The translator gets mapping entries for the IPv6 destination address and the IPv6 source address from the mapper in

the same way as before. Then the translator translates the IPv6 packet into an IPv4 packet and tosses it up to the application.

Diagram 1.5 illustrates the action described above:

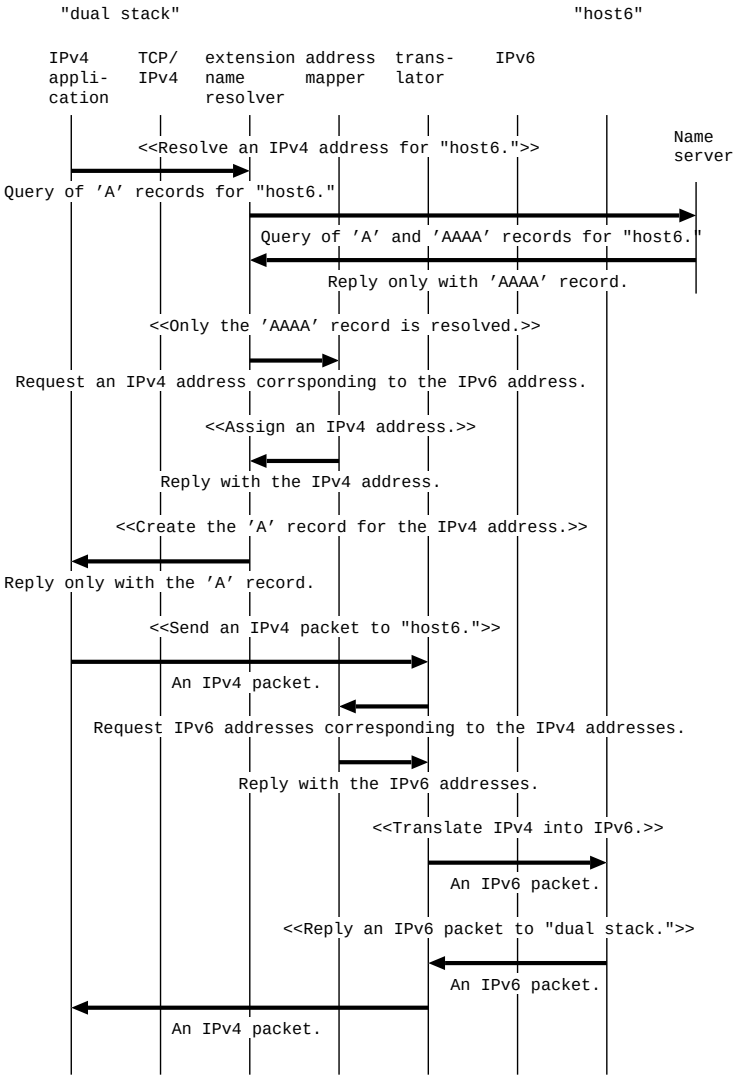


図 1.5: Originator behavior

Recipient behavior

This subsection describes the recipient behavior of "dual stack." The communication is triggered by "host6."

"host6" resolves the AAAA record for "dual stack" through its name server, and it then sends an IPv6 packet to the resolved IPv6 address.

The IPv6 packet reaches the translator of "dual stack." The translator tries translating the IPv6 packet into an IPv4 packet but does not know how to translate the IPv6 destination address and the IPv6 source address. So the translator requests the mapper to provide mapping entries for them.

The mapper checks its mapping table with each of them and finds a mapping entry for the IPv6 destination address.

NOTE: The mapper will register its own IPv4 address and IPv6 address into the table beforehand.

But there is not a mapping entry for the IPv6 source address, so the mapper selects an IPv4 address out of the spool for it, and then returns the IPv4 destination address and the IPv4 source address to the translator.

NOTE: See subsection 1.4.5 about influence on other hosts caused by the assigned IPv4 address.

The translator translates the IPv6 packet into an IPv4 packet and tosses it up to the application.

The application sends a new IPv4 packet to "host6."

The following behavior is the same as that described in subsection 3.1.

Diagram 1.6 illustrates the action described above:

1.4.5 Considerations

This section considers some issues with the proposed dual stack hosts.

IP conversion

In common with NAT [47], IP conversion needs to translate IP addresses embedded in application layer protocols, which are typically found in FTP [108]. So it is hard to translate all such applications completely.

IPv4 address spool and mapping table

The spool, for example, consists of private addresses [110]. So a large address space can be used for the spool. Nonetheless, IPv4 addresses in the spool may be exhausted and cannot be assigned to IPv6 target hosts if the host communicates with great many other IPv6 hosts and the mapper never frees entries registered into the mapping table once. To solve the problem, for example, it is desirable for the mapper to free the oldest entry in the mapping table and re-use the IPv4 address for creating a new entry.

Internally assigned IPv4 addresses

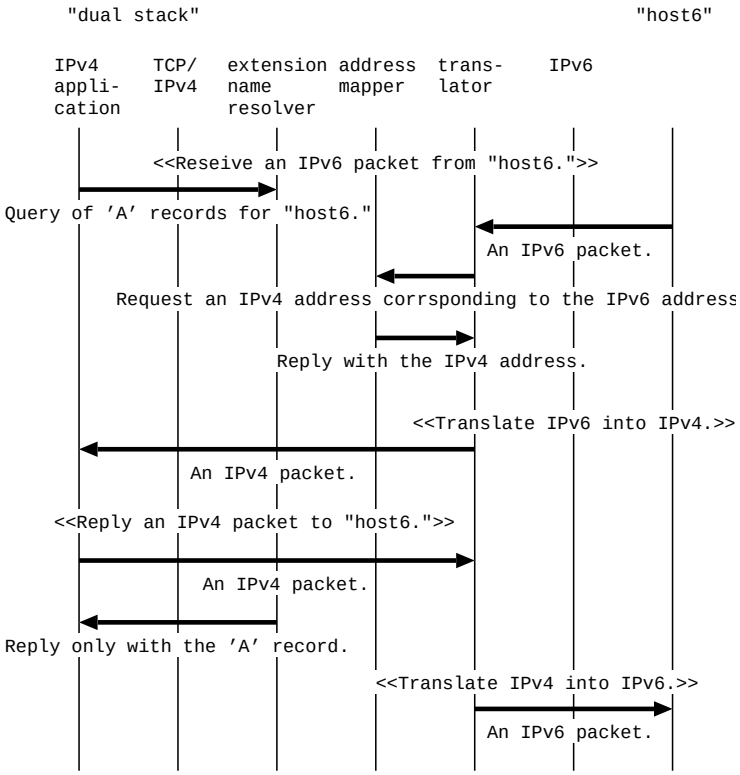


図 1.6: Recipient behavior

IPv4 addresses, which are internally assigned to IPv6 target hosts out of the spool, never flow out from the host, and so do not negatively affect other hosts.

1.5 SOCKS64

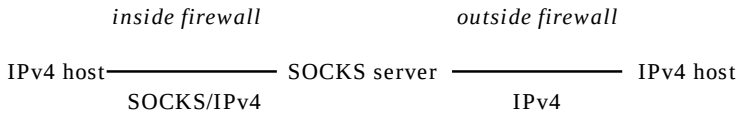
1.5.1 Abstract

This memo describes an IPv4-IPv6 intercommunication method based on the SOCKS5 protocol. The method is called SOCKS64, and was first announced in the 40th IETF meeting at Washington DC in December 1997 as one of three translators being developed in the WIDE Project. SOCKS64 is a gateway system that accepts SOCKS5 connections from IPv4 hosts and relays it to IPv4/IPv6 hosts using proper protocols. We have designed and implemented a prototype SOCKS64 system and have been testing the prototype in many environments. Also we are distributing the prototype as a KAME distribution package. This article describes the SOCKS64; its principle, implementation, experimental results and comparison to other IPv4-IPv6 intercommunication methods.

1.5.2 Principle

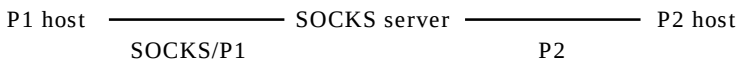
In this section, we describe the SOCKS system briefly and figure out a basic idea to use it as an intercommunication gateway system for different protocols.

The SOCKS protocol has been developed as a firewall gateway protocol that provides hosts in a firewall with a relay service of transport layer (TCP) data to outside of the firewall. A SOCKS server, placed in a border of the firewall, accepts a SOCKS connection from a host inside the firewall, makes a new connection to a host outside the firewall, and relays the transport layer data between these two connections.



Once the SOCKS scheme is understood, it is easy to imagine to connect different protocols in the similar way.

First, a host with a protocol-1 (P1) connects to a SOCKS server using the SOCKS protocol over P1. Next the SOCKS server makes a connection to the destination host with a protocol-2 (P2) using P2. Once the connections are established, the SOCKS server starts to relay the transport layer data between these connections.

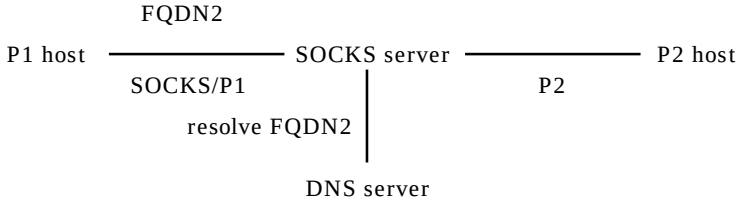


It seems to be a simple way to connect two protocols, though there is an big problem to complete such a scheme, an addressing system problem.

Any protocol has its own addressing system. The P1 host should know an address of the P2 host, when it wants to connect to. Generally, the P1 host has no way to get the P2 address because it does not know the P2 addressing system. To solve this problem, there should be an addressing system that is recognized by both protocols.

Fortunately, in the world of Internet Protocol, we have the Domain Name System[DNS1, DNS2] and the Fully Qualified Domain Name (FQDN) as a rigid solution. Using DNS, the P1 host can address the P2 host using FQDN2, and the P2 host can address the P1 host using FQDN1. In this case, the story goes like the following.

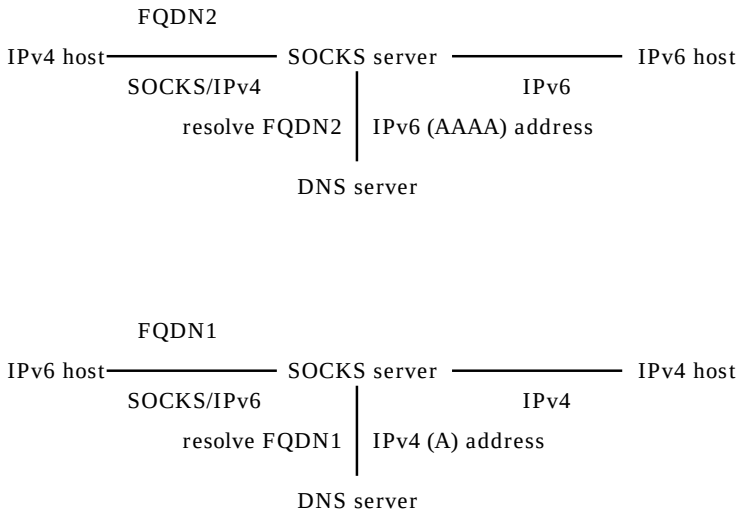
First, the P1 host asks the SOCKS server to connect to the P2 host specifying FQDN2 as the address. Then the SOCKS server asks the DNS server for the actual address of FQDN2 and gets the P2 address. So the SOCKS server can make a connection with the P2 host, and relay the transport layer data between two connections.



Using this scheme, a SOCKS server that is constructed on an IPv4-IPv6 dual stack host can act as such a transport relay gateway, if the SOCKS protocol has a capability to specify connecting hosts by FQDN addresses.

Originally, SOCKS was for IPv4 and was not made to handle other protocols than IPv4. The SOCKS protocol versions before SOCKS5 could treat only the IPv4 address, so it could not relay connections between hosts with different protocols. But when the SOCKS5 protocol[77] has introduced the FQDN address type, it became possible to use the SOCKS protocol for an intercommunication gateway of different network layer protocols, IPv4 and IPv6.

In the case an IPv6 host wants to connect to an IPv4 host, a completely symmetric scheme can be used.



1.5.3 Implementation

Based on the principle described above, we have implemented a prototype SOCKS64 server based on the SOCKS5 reference code [1]. The SOCKS64 prototype is distributed freely[2].

In this section, we describe an implementation of the SOCKS64 server based on our experiences. Current implementation of the SOCKS64

prototype runs on BSD/OS 3.1 with the KAME IPv6 protocol stack [3]. There is no modification in the client library for the IPv4 hosts. As for the client library for the IPv6 hosts, the new functions such as `getaddrinfo()` should be provided. However, our current implementation does not provide such new functions.

Implementing the SOCKS64 server, there is no fundamental problem. Our implementation of the SOCKS64 server consists of two major modifications to the SOCKS5 reference code. One is to make the SOCKS5 server IPv6 capable, and the other is to add a framework which enables the SOCKS5 server to do application specific processings.

The former modification includes the changes in "struct sockaddr", and making the configuration file parser to be able to read IPv6 addresses as specified in RFC2373[63].

The other modification is needed because some applications use IP addresses in their protocols. For example, FTP protocol[108] specifies that an FTP server and an FTP client exchange the IP addresses in text format. So it is required to realize a processing for FTP, converting the PORT command into the LPRT command, the PASV command into LPSV

[1] `http://www.socks.nec.com/`

[2] `ftp://ftp.kame.net/pub/kame/misc/socks64-v10r8-19990118.tgz`

[3] `http://www.kame.net/`

command, and translating between the IPv4 address and the IPv6 address. The LPRT command and the LPSV command are described in RFC1639[105].

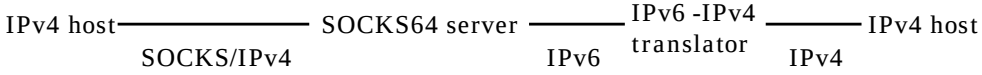
1.5.4 Experiments

We have been testing the SOCKS64 server in many environments. Since July 1998, we have been running the SOCKS64 server in between the 6bone and the internal network of Fujitsu Ltd., Japan. Computers in Fujitsu's internal network already have been "socksified". So users can connect to 6bone hosts only selecting our SOCKS64 server as their SOCKS gateway.

For an interoperability test, we have been providing a SOCKS64 server in the WIDE Project [4] Camp, where about 200 Internet researchers attend. A temporary network called "camp-net" is constructed at the camp for a variety of experiments and provides services to the attendees. More than hundred hosts are connected to the camp-net.

In these environments, we have tested many types of SOCKS5 clients, including the SocksCap32 [5] on Windows and a variety of UNIX based SOCKS libraries, and confirmed any clients successfully connected to the IPv6 hosts using telnet, ftp, http, ssh.

Also we have tested much complicated interconnection of IPv4 and IPv6. At the WIDE camp held in September 1998, we constructed a IPv4 over IPv6 tunnel using the SOCKS64 and the NAT based translator. In this system, the SOCKS5 clients could connect to the IPv4 hosts with no problem through the IPv6 networks.



Concluding, SOCKS64 provides an easy and sure way to let IPv4 hosts connect to IPv6 hosts.

1.5.5 Considerations

In this section, we compare the SOCKS64 method to other IPv4-IPv6 interconnection methods.

The common way being developed is the "translator" based on the Network Address Translation (NAT) technology. The translator translates IPv4 packets and IPv6 packets one by one. To solve the addressing system problem, the translator has an association table

[4] http://www.v6.wide.ad.jp/

[5] http://www.socks.nec.com/sockscap.html

of IPv4 and IPv6 addresses, and uses these addresses to translate packets. This approach requires an extension of DNS to manage the address association table. SOCKS64 does not need such changes.

A shortcoming of the translator is a transport layer data processing. Because the translator works in the network layer, it has some difficulties when the transport layer data is required to be changed. The FTP addresses embedded in the transport layer data is an obvious example.

The application-level gateways can change the transport layer data naturally. FAITH, distributed in the KAME distribution package, is an example of such approach. FAITH does not require client hosts to change their software. But it requires special DNS treatment as same as the NAT based translator.

On the other hand, the biggest shortcoming of SOCKS64 is that every clients should be "socksified". It should be a bothering work for network managers to support SOCKS users.

Here is the summary of comparison described above.

	Implementation Layer	DNS Change	Address Table	Client Library
NAT	Network	needed	needed	not needed
FAITH	Application	needed	needed	not needed
SOCKS64	Application	not needed	not needed	needed

1.5.6 Conclusion

The SOCKS approach provides a reasonable way to construct an IPv4-IPv6 intercommunication gateway. Especially, many firewall users who are already "SOCKS ready" can communicate with IPv6 hosts with no special care. Firewall managers also need no extra care adopting SOCKS64, because SOCKS64 preserves all security features the original SOCKS has.

1.6 An overview of the KAME network software

1.6.1 Abstract

The KAME Project is a joint effort by seven companies in Japan to develop a referential implementation of advanced networking protocols such as IPv6 and IP security. The products are distributed as free software and are widely used in the 6bone. They are based on BSD variants and are added several improvements to the original logic, which cannot

handle advanced protocols. This paper describes such new mechanisms as efficient header processing, loop prevention for tunneling, and mobility support for IPv6 plug and play.

1.6.2 Introduction

With the rapid growth of the Internet, more and more significant and unexpected problems have appeared. In response to some of these problems, IP security(IPsec) was developed as a security mechanism while IPv6 was designed to resolve the exhaustion of the IPv4 address space. As BSD variants promoted to deploy TCP/IPv4, a free referential implementation of IPsec and IPv6 is necessary for their wide and early deployment.

For this purpose, the KAME Project was initiated in April 1998 in Japan by seven Japanese companies co-operating in the WIDE (Widely Integrated Distributed Environment) Project. It aimed to provide free, working, and "specification conformant" code based on BSD variants.

The logic of the network code of the current BSD variants are unsuitable for the new protocols. For instance, using the old logic, efficient processing of IPv6 extension headers is not possible. Also, it is time-consuming to determine if an incoming packet should be accepted or forwarded. To overcome these difficulties, the KAME Project has explored some original approaches in its implementation.

This paper describes the implementation techniques developed by the KAME Project. It is organized as follows: Section 2 briefly describes KAME Project itself. Section 3 explains the characteristics of KAME implementation. We describe future plans in section 4 and conclude in section 5.

1.6.3 The KAME Project

The WIDE Project has organized the IPv6 working group and has been deeply involved in IPv6 since 1995. More than one hundred researchers from universities and companies have committed themselves to the working group. The working group has developed nine independent IPv6 implementations, and the group has tested the interoperability between them through its testbed, called the WIDE 6bone [136].

Such heterogeneity was important in the early stages of IPv6 development since it helped to improve each implementation and to find out problems of specification. As IPv6 has been deployed, heterogeneity has performed its role, and instead, efficient development of products and early deployment of the products have become important.

Thus the WIDE IPv6 working group organized the KAME Project. Eight core developers from seven companies came together to develop a single more efficient referential implementation of the advanced networking protocols. The project chose several BSD variants

as its platform: FreeBSD, BSD/OS, and NetBSD. At the start, the product of the project was provided in a form of patches for the original operating systems, but the project also aimed to incorporate the product into official releases of the operating systems.

To promote these advanced protocols, the product is freely distributed via the project web site (`http://www.kame.net/`). The product is distributed in the following three ways:

- A SNAP release for active users such as researchers is released once a week. It includes many experimental items, and thus may or may not be stable.
- A STABLE release for broader public is made twice a month. Since it is intended to be a stable snapshot, experimental items are not included in a STABLE release.
- An official release is called RELEASE.

The KAME product is actually used in the worldwide 6bone [6]. It has also been tested for interoperability and conformity at the InterOperability Laboratory at the University of New Hampshire. The KAME Project is planning to start its own group to evaluate the product. Official releases will be made after quality assurance testing by the group.

1.6.4 Implementation Characteristics

This section describes some of KAME's remarkable network software characteristics including basic specifications and neighbor discovery. Loop prevention is related to IPsec as well as to IPv6.

IPv6 Extension Headers

One remarkable feature of IPv6 is its flexible extension headers [42] combined with the constant length IPv6 header. The limitations of each extension header's length and of the number of extension headers are looser than those of IPv4 options. Thanks to this flexibility, we can try to introduce a new optional function as a separated extension header or as an option of some existing options header.

Traditional BSD variants store an incoming packet in one or more mbufs [132]. There are two types of mbuf. In this paper, we refer to them as an internal mbuf and an external mbuf, respectively. By default, an internal mbuf can contain only 100 to 108 bytes of data. An external mbuf has external storage, called a cluster, whose size is 2048 bytes by default. If an incoming packet fits in a single internal mbuf, the network driver simply stores the packet in the mbuf. Otherwise, the network driver stores the packet into mbufs one of two ways as follows:

[6] `http://www.6bone.net/`

- The packet is stored in a single external mbuf. If the packet does not fit in the external mbuf, the driver divides the packet into fragments, stores each fragment into an external mbuf, and makes a chain of the mbufs.

- If the packet fits in two internal mbufs, the driver divides the packet into two fragments, stores each fragment into an internal mbuf, and makes a chain of the internal mbufs. Otherwise, the driver stores the packet in the same as above, that is, stores it in one or more external mbufs.

In the latter mode, there is a possibility that the first mbuf cannot contain all the headers from the IPv6 header to an upper layer protocol header(e.g. a TCP header). Moreover, a header may be divided and spread over two or more mbufs(Figure 1.7).

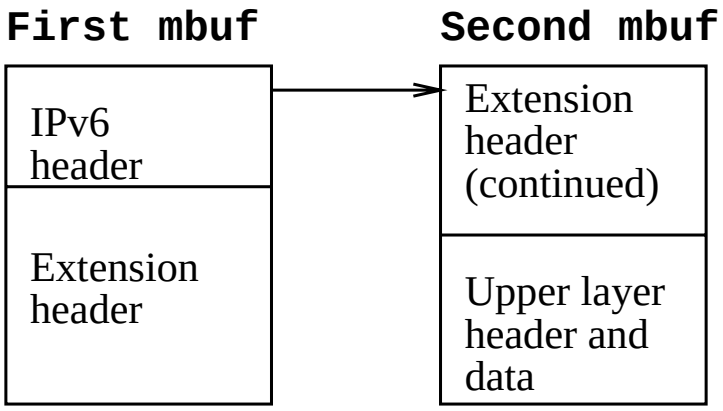


図 1.7: An extension header may be spread over two mbufs

BSD variants usually expect that the network layer header and the transport layer header of an incoming packet lie on a contiguous memory space. If this is not the case, they perform the "pullup" operation, which copies specified length of data into a contiguous space in a single mbuf. If there are some IPv4 options, they strip the options before the operation.

This was a good idea when memory resource was a scarce commodity, but the overheads of the copy operation are heavy. In addition, the pullup operation is not suitable for IPv6 environments from another point of view. We must retain all the intermediate extension headers until the input operation terminates, since if we encounter an error while processing an incoming packet, we have to return an ICMPv6 error message including as much of the offending packet as will fit [39]. However, if we follow the original convention of BSD variants, some intermediate headers may have been stripped or even discarded before the pullup operation as mentioned above.

Since the cost of memory continues to rapidly decline, processor efficiency should be favored over memory efficiency. From this perspective, we require network drivers to store all the headers in a contiguous space in order to avoid pullup operations. That is, a network driver that is suitable for the KAME network software is required to store an incoming packet in the former of the two modes mentioned above.

To satisfy this requirement, we can write an input routine for a header more simply than before. Each input routine first determines if the requirement is satisfied, and if not, it discards the packet. Otherwise, the routine gets the beginning of the header by the head of the incoming packet and the offset to the header. Then the routine can process the header and, if necessary, the data following the header without any copies or deletions of other headers.

Though we have to rewrite a network driver if it does not satisfy this requirement, most recent network drivers do not have to be rewritten. This is a reflection of the current fashion wherein code simplicity is regarded as more important than memory efficiency.

1.6.5 An Efficient Method to Examine Destination Addresses

For an incoming IPv4 packet, traditional BSD variants first determine whether the packet is destined for the node or not. If the packet is destined for the node, the receiving node passes the packet to an appropriate transport layer. Otherwise, the node tries to forward the packet if it is configured to act as a router.

This determination is usually done by comparing the destination address of the incoming packet with each address assigned to the interfaces of the node. Since the comparison linearly examines the addresses, in the worst case, the node has to examine all the addresses. However, this is not very harmful to IPv4 environments because even an IPv4 router does not have many addresses. Moreover, it is a relatively inexpensive task for today's computers to compare IPv4 addresses since the addresses are 32-bit integers.

In contrast, a linear comparison has more significant effect on IPv6 environments. IPv6 capable nodes may assign multiple IPv6 addresses on a single interface, and consequently, routers may have numerous IPv6 addresses. Also, because today's typical computers are based on 32-bit(or at most 64-bit) architectures, it is fairly time-consuming to compare IPv6 addresses, which are 128-bit integers.

Therefore, we have adopted a different scheme for deciding whether an incoming IPv6 packet should be accepted or forwarded. The kernel first looks up the radix routing table [114] to resolve the next hop of an incoming packet and the outgoing interface to the next hop. If the interface is a loopback interface (usually called lo0 in BSD variants), it accepts the packet. Otherwise, it uses the next hop information to forward the packet.

For a router, our method is efficient since the number of packets to be forwarded is much

higher than accepted packets, and there is no additional cost for forwarded packets. For a host, if an incoming packet is destined for the host itself, our method needs a few of bit tests (to reach a leaf node of the radix tree) and a comparison of two IPv6 addresses[14]. Figure 1.8 roughly depicts the process. The cost of the comparison can be ignored because it is definitely necessary at least once for any method. Also, it can be safely assumed that the radix tree for a host is not complex since a host usually has only a few host routes in addition to the default route, and, consequently, the cost of the tests can be considered relatively low. Even when the host accidentally receives a packet destined for another node, the host can quickly detect it thanks to the simplicity of its radix tree.

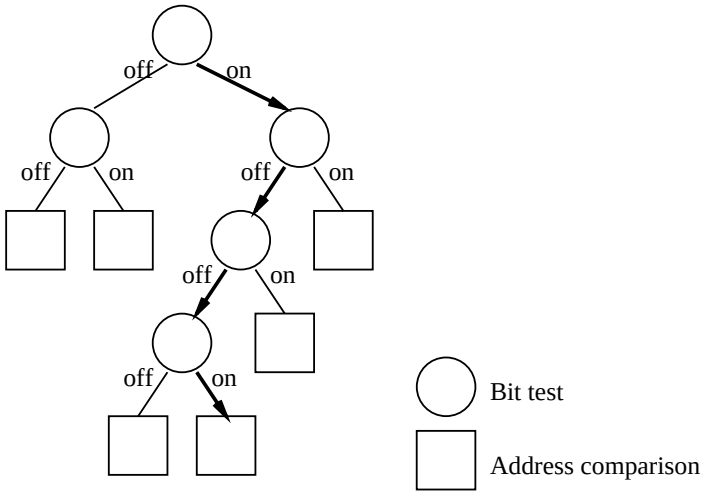


図 1.8: The cost of radix tree lookup is bit tests on the internal nodes and a comparison of addresses at the leaf node

Another improvement intended to avoid linear search is to use a hash technique. A node calculates in advance a hash value of each address assigned to it, then makes a hash table. Each entry of the table is a list of addresses that have a same hash value. For an incoming packet, the node calculates the hash value for the packet's destination address, and compares the address to each address that belongs to the list for the hash value. If we have a good hash algorithm, the comparison is fast enough. In some cases, it will be faster than the radix tree-based comparison. However, if a node is acting as a router, our approach is more efficient.

Loop Prevention for Tunneling Techniques

There are some notions of tunneling in advanced internetworking such as IPv6 and IPsec. For example, IPv6 over IPv4 tunneling[38] is an essential technique during the transition

period from IPv4 to IPv6. The IPsec tunnel mode [75] is typically used to construct a VPN(Virtual Private Network).

To implement such tunnels, it is necessary to encapsulate one packet inside another. The implementation may use a pseudo network interface, and the output function associated to the interface receives a packet from a network layer (e.g. IPv4 layer), encrypts and/or authenticates the packet if necessary, and encapsulates it in an outer header. The input function, in contrast, decapsulates an incoming packet, decrypts and/or authenticates the packet, and passes it to a corresponding network layer. When a route for a destination is configured to the pseudo interface, all packets to the destination will be transferred through the tunnel associated with the pseudo interface.

Such an approach may, however, fall into an infinite loop of header creation. An infinite loop is essentially a sequence of processes which eventually reaches an already passed point. Since an output to a pseudo network interface may cause an output to the same interface, the sequence of the outputs has a possibility of an infinite loop.

As a simple example, suppose that we want to construct an IPv4 IPsec tunnel for a destination D and that we, perhaps mistakenly, configure the pseudo interface, for example I, with the same destination D. As we already mentioned, a route for destination D is set up to I. When we try to send a packet to D, the packet is passed to I according to the routing table, encrypted and/or authenticated in I, and encapsulated into a new header, whose destination address is also D. Then the encapsulated packet is sent to the IPv4 output routine, which sends the packet to I again. Thus we enter an infinite loop (Figure 1.9).

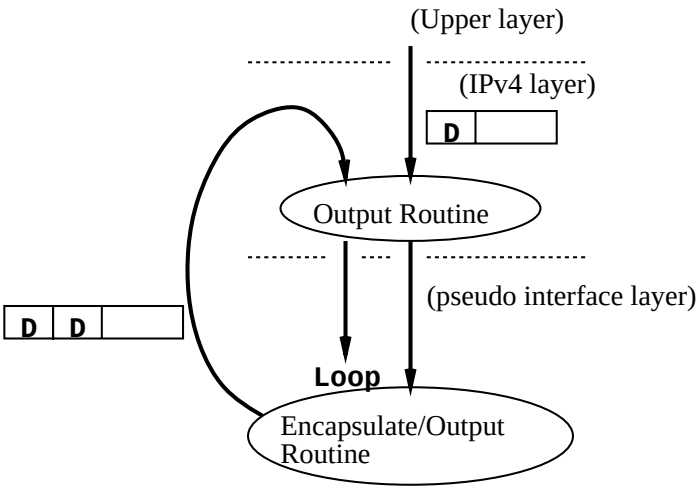


図 1.9: Example of an infinite loop

To deal with this problem, we implemented IPsec tunnel mode not as an interface, but as a separate routine which is called from a network layer output function only once (Figure 1.10).

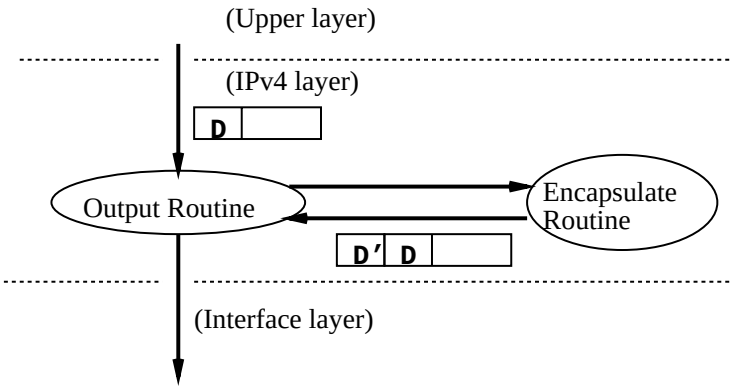


図 1.10: The IPsec encapsulation routine is called only once to prevent a loop

As a result, an IPsec tunnel can be constructed only once in a single node, though this is not a strict restriction because IPsec tunnels are not created more than once in a practical configuration.

An infinite loop may occur when constructing a tunnel which encapsulates a network protocol into a different network protocol, such as an IPv6 over IPv4 tunnel. For example, suppose that we construct two tunnels. One is an IPv6 over IPv4 tunnel, and the other is an IPv4 over IPv6 tunnel. Also suppose that the IPv6 over IPv4 tunnel encapsulates a packet to an IPv6 address D6 into a packet to an IPv4 address D4, and that the IPv4 over IPv6 tunnel encapsulates a packet to D4 into a packet to D6. If we try to send a packet to D6 we encounter a loop which infinitely constructs the two tunnels one after another (Figure 1.11).

However, such a combination of tunnels like the above example does not appear in a typical configuration. Hence we implemented a generic framework consisting of a pseudo interface and used it to construct IPv6 over IPv4 tunnels. To avoid possible loops, we introduced a counter in the output function for the interface, which is incremented each time the function is recursively called. If the counter reaches some limit, the function discards the packet, records that it detects a loop, and returns an error to the user. The flexibility of the pseudo interface is decreased to some extent by the use of this trick, but in this instance, we believe robustness should take priority over flexibility.

IPv6 Plug and Play for Mobile Environments

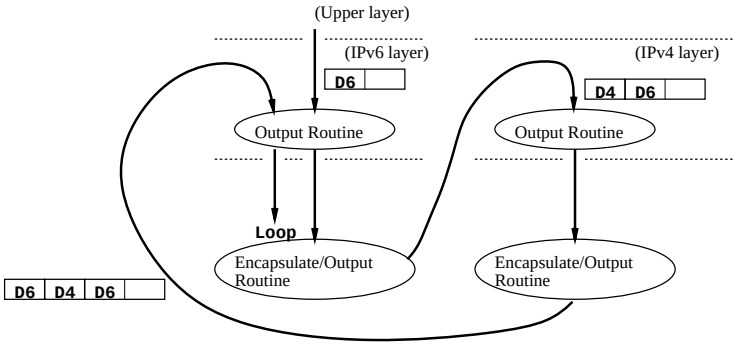


図 1.11: An infinite loop consisted of two tunnels

Plug and Play based on Neighbor Discover Protocol [94][121] is one of IPv6's major improvements. When a node attaches to an IPv6 network, the node can automatically configure its IPv6 address(es) and discover one or more available routers. Moreover, the configuration process needs no state machine in the server. Network managers do not have to configure a special server such as a DHCP server. They only have to configure network prefixes for routers, which is eventually necessary to achieve global connectivity.

Once a router is configured, it periodically advertises the configured network prefixes to each link attached to the router. Each host on the network receives the advertisements and configures itself using the prefix(es) contained in the advertisements. A host also regards the sender of the advertisement as a default router, thus achieving connectivity to the global IPv6 Internet.

The specification defines two types of lifetime for an advertised prefix. One is preferred lifetime(default value is 7 days) and the other is valid lifetime(default value is 30 days). They are used to invalidate prefixes smoothly. If the preferred lifetime of a prefix has expired, the addresses generated by the prefix should not be used as a connection's source address unless it is already established. If the valid lifetime of a prefix has expired, the addresses generated by the prefix becomes completely invalid; that is, the addresses must not be used as a source address for any connections. This mechanism is useful, for example, when a site is renumbering its addresses.

The default values of the lifetimes are, however, relatively long for mobile nodes. Consider this problematic case as an example. Suppose that there are two networks, N1 and N2, and that prefixes P1 and P2 are advertised in N1 and N2, respectively. Now consider that a mobile node M moves from N1 to N2. When M attached to N1, an IPv6 address P1:M was configured for M. Then M moves to N2 and configures a new address P2:M. If the movement happens quickly, which is usually the case, the lifetimes of the old prefix P1 do

not expire and the old address P1:M is, as a result, still valid.

Let us assume that M sends a packet to an off-link node D through a router R in N2. There are two possible addresses as the source address of the packet; P1:M and P2:M. But if the former is used and D tries to send a response to it, D will send the response to P1:M. Since the prefix P1 lies on the network N1, M is not able to receive the response unless the host route for P1:M is fully advertised. Figure 1.12 depicts this situation.

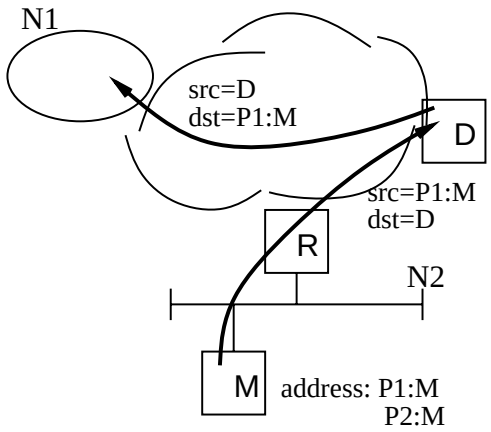


図 1.12: A mobile node can send a packet, but cannot receive responses

There is another problem if M tries to send a packet to a node D in N1 whose address is P1:D. In this case, since D and M have the same prefix P1, M will regard D as on-link and try to send the packet to it directly instead of sending to the neighboring router R(Figure 1.13). But the packet will not reach D because in reality D is off-link.

One simple solution to these problems is to force the node to discard the old prefix and the old address after moving. But this approach may cause an unexpected deletion of prefixes and addresses that should remain valid. For instance, suppose that there is a failure of the router on a network while a laptop computer is suspended. When the laptop's operation is resumed, it should discard the prefix and the address that were advertised before suspension, since there is a possibility that the laptop has moved from the network. The laptop, however, cannot get a prefix any more because the router has already stopped; it therefore fails to communicate with any nodes, even with those within the network.

Thus we implemented a mobility support that is a bit more complicated. We introduced a data structure for each prefix, which has a list of routers that advertise the prefix. Note that the list may be empty when, for example, all routers that advertise the prefix are unreachable. Then we defined two states for a prefix, attached and detached. A prefix is called attached if its router list is not empty or if all the prefixes including the prefix do not

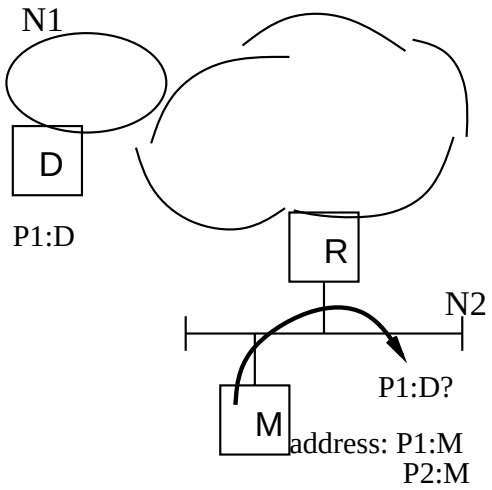


図 1.13: A mobile node regards an off-link node as on-link by mistake

have a non-empty router list. Otherwise, the prefix is called detached. A detached prefix is not regarded as on-link and the address derived from the prefix is not used as a source address even if the lifetimes of the prefix are not expired.

Based on the above model, we implemented a mechanism in the kernel to control the state of each advertised prefix. For instance, if a router turns out to be unreachable, which can quickly be detected using IPv6 Neighbor Unreachability Detection, the kernel automatically changes the state of each prefix advertised by the router.

To return to the above problems, since the prefix P1 of the mobile node M of Figure 6 is detached after detecting unreachability of the router in N1, the old address P1:M will be never used as a source address. Also, since the detached prefix P1 is no longer regarded as on-link, M will pass a packet to a neighboring router when the packet is destined for a node in N1. Now let us consider the case of the failure of the router. If the router R of Figure 6 stops, both P1 and P2 are attached since there is no other prefix that has an associated router. In this case, M can only communicate with nodes in N2, and there is no problem since M can use the addresses derived from the attached prefixes.

Portability

It is important to ensure the source-level portability of applications on various operating systems. Portability encourages people to develop a variety of applications, and consequently we have many opportunities to test and improve inter-operability, which is also an important notion on the Internet.

Application Programming Interfaces (APIs) play a key role to ensure portability. There

are two types of API defined by IETF: The first is the basic API[59], which defines some library functions, data structure, and macros for developing typical TCP and UDP applications. The other is the advanced API [117], which is designed to develop and support "advanced" applications such as routing daemons and network management tools. Interfaces designed to use IPv6 extension headers are also defined in the API.

Since we attach importance to portability, we have quickly adopted the latest version of these APIs. We have also tried to use the APIs both in the kernel and in userland applications. For example, all our "advanced" applications such as routing daemons, a router advertisement daemon, ping, and traceroute use the advanced API.

Adopting the latest specification of such APIs does not necessarily provide portability, and even may cause confusion due to version mismatches. This is because the specification tends to change in the early stages of its standardization. We believe, however, that in order to encourage the early standardization and deployment of the APIs, it is more important to actively introduce the latest specifications than to stick to short-term portability.

1.6.6 Future Plans for New Development

IPv6 and IPsec have already become fairly standardized, but there are still some topics that are neither well-documented nor fully standardized.

It is difficult for network managers to renumber their sites, and hence they rarely change their network providers. Router renumbering[41] is one of the key techniques needed to remedy this situation. Though its specification has not been fully standardized, we are now implementing it experimentally and are planning to run it on the WIDE 6bone.

IPv6 Multicast routing is also in the early stages of deployment. Some documentation for the IPv6 PIM[60][50] exists, but there are few instances of implementation and interoperability between different implementations is not fully established. So far, we have implemented PIM dense mode for IPv6 and have confirmed that it works to some extent. Next, we will have to test it in practical applications and check its interoperability with other implementations.

1.6.7 Conclusion

The WIDE IPv6 working group organized the KAME Project in order to develop a referential implementation of advanced networking protocols for BSD variants and to promote the protocols through their implementation.

Because some of the logic used in BSD variants was not well-suited to the advanced protocols, we explored different approaches to implementing the KAME network software; we implemented incoming IPv6 packets processing in an efficiency-conscious manner. We

prevented infinite IPsec header creation loops by restricting the header construction to a single instance. Infinite loops in other tunnels such as IPv6 over IPv4 tunnels, meanwhile, was avoided by introducing an upper limit for nesting. IPv6 neighbor discovery was implemented so that it would be more compatible with mobile stations.

In the future, we plan to continue the project with special emphasis on such areas as the implementation of more advanced technologies such as router renumbering and IPv6 multicast routing.

1.7　年間を通じての活動

この節では v6 分科会が 1998 年度を通じて主催、および参加した活動内容をまとめる。

8 月 17 日 ～ 8 月 21 日 第 5 回 相互接続実験 (ニューハンプシャ大学)

8 月 24 日 ～ 8 月 28 日 第 42 回 IETF(シカゴ)

10 月 27 日 ～ 10 月 30 日 IPsec 相互接続実験 (ニューヨーク州)

12 月 7 日 ～ 12 月 11 日 第 43 回 IETF(オーランド)

12 月 15 日 6bone BOF(IP ミーティング'98)

1 月 21 日 マルチホーム BOF(東京大学)

3 月 15 日 ～ 3 月 19 日 第 44 回 IETF(ミネアポリス)