第8部

次世代インターネットプロトコル

第 1 章　v6 分科会

v6 分科会は、IPv6 と IPsec に関する研究に取り組んでいる。本章では、1999 年度の v6 分科会の成果報告として、主に IETF での活動について述べる。

1.1 概要

v6 分科会のメンバーは、IETF の ipngwg 分科会、および ngtrans 分科会において積極的に発表している。最近では、議長から依頼されて発表することも多くなってきた。以下に発表用件についてまとめる。

- 第 45 回 IETF、ノルウェー、オスロ、1999 年 7 月 11 日–16 日
  - ipngwg
    - TAHI Project: Verification technologies for IPv6 、岡部宣夫
    - IPv6 Multihoming – RFC2260 approach、萩野純一郎
  - ngtrans
    - IPv6 Network in WIDE CAMP、関谷勇司
    - BIS: Bump-in-the-Stack Technique、土屋一暁
- IETF ipngwg 分科会中間ミーティング、田町、東京都、1999 年 9 月 29 日–10 月 1 日
  - 本会議
    - ICANN IPv6 status、村井純
    - IPv6 Multihoming – router only mechanisms、萩野純一郎
    - An Extension of Format for IPv6 Scoped Addresses、神明達哉
  - 日本セッション
    - Recent Activity of the KAME Project、神明達哉
    - Hitachi IPv6 activity and GR2000、角川宗近
    - IIJ IPv6 trial service、萩野純一郎
    - 1st TAHI IPv6 Interoperability test、宮田宏
    - WIDE 6bone、長橋賢吾
    - IPv6 install convention in Japan、許 先明 (HEO SeonMeyong)
- 第 46 回 IETF、アメリカ、ワシントン DC、1999 年 11 月 7 日–2 日
  - ipngwg
    - An Extension of Format for IPv6 Scoped Addresses、神明達哉
    - Tokyo meeting network configuration、萩野純一郎
    - TAHI Project: Interoperability Test Report、宮田宏
  - ngtrans
    - BIS: Bump-in-the-Stack Technique、土屋一暁
- 第 47 回 IETF、オーストラリア、アデレード、2000 年 3 月 26 日–31 日
  - ipngwg
    - Resolving addresses of the root DNS servers、山本和彦
    - Possible abuse against IPv6 transition technologies、萩野純一郎
    - An Extension of Format for IPv6 Scoped Addresses、神明達哉
    - IPv6 IPsec test in connectathon2000、星野浩志
  - ngtrans
    - Overview of Transition Techniques for IPv6-only to Talk to IPv4 -only Communication 、山本和彦
    - An IPv6-to-IPv4 transport relay translator、萩野純一郎
    - Possible abuse against IPv6 transition technologies、萩野純一郎
    - Root DNS for IPv6 transport、加藤朗

- ipsec
  - TAHI IPsec test suites、星野浩志

ここで、上記の IETF ipngwg 分科会中間ミーティングについて説明する。1999 年 9 月 29 日から 10 月 1 日に、WIDE/KAME プロジェクトがホスト役となり、IETF ipngwg 分科会中間ミーティングを開催した。IETF に関連するミーティングを日本で開催するのは初めてである。テーマは、IPv6 環境におけるマルチホーミング。91 人 (満席) の出席者を得て、内容としても成功に終わった。10 月 1 日の午後は日本での IPv6 の活動を外国人に説明する時間を設け、ipngwg 分科会議長などに理解を深めて頂いた。詳しくは、以下のページを参照して頂きたい。

```
http://www.wide.ad.jp/events/
            199909.ipng-interim/
```

上記のほとんどは、RFC や Internet-Draft について発表したものである。これらの文献をまとめる形で、本稿は次のように構成される。

- Bump-in-the-Stack に関する RFC (Informational)
- アドレスの悪用に関する Internet-Draft
- トランスポートリレーに基づいたトランスレータに関する Internet-Draft
- 新しいグループ通信に関する Internet-Draft
- マルチホームに関する Internet-Draft
- トランスレータの分類に関する Internet-Draft
- スコープを持つアドレスの取り扱いに関する Internet-Draft

1.2 Dual Stack Hosts using the "Bump-In-the-Stack" Technique (BIS)

1.2.1 Introduction

RFC1933 [56] specifies transition mechanisms, including dual stack and tunneling, for the initial stage. Hosts and routers with the transition mechanisms are also developed. But there are few applications for IPv6 [125] as compared with IPv4 [57] in which a great number of applications are available. In order to advance the transition smoothly, it is highly desirable to make the availability of IPv6 applications increase to the same level as IPv4. Unfortunately, however, 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" [58] in the IP security area. The technique inserts modules, which snoop data flowing between a TCP/IPv4 module and network card driver modules and translate IPv4 into IPv6 and vice versa, into the hosts, and makes them self-translators. When they communicate with the other IPv6 hosts, pooled IPv4 addresses are assigned to the IPv6 hosts internally, but the IPv4 addresses never flow out from them. Moreover, since the assignment is automatically carried out using DNS protocol, users do

not need to know whether target hosts are IPv6 ones. That is, this allows them to communicate with other IPv6 hosts using existing IPv4 applications; thus it seems as if they were dual stack hosts with applications for both IPv4 and IPv6. So they can expand the territory of dual stack hosts. Furthermore they can co-exist with other translators because their roles are different.

This memo uses the words defined in [57], [125], and [56].

1.2.2 Components

Dual stack hosts defined in RFC1933 [56] 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. They are a translator, an extension name resolver and an address mapper.

Figure 1.1 illustrates the structure of the host in which they are installed.

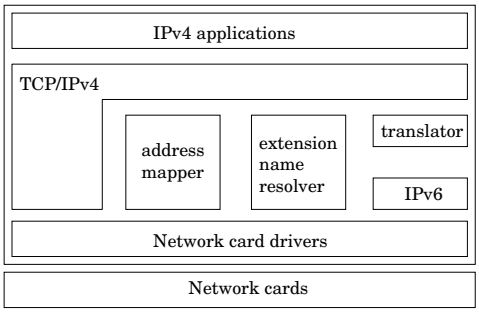


図 1.1 Structure of the proposed dual stack host

Translator

It translates IPv4 into IPv6 and vice versa using the IP conversion mechanism defined in [59].

When receiving IPv4 packets from IPv4 applications, it 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

It returns a "proper" answer in response to the IPv4 application's request.

The application typically sends a query to a name server to resolve `A` records for the target host name. It 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 as is. In the case, there is no need for the IP conversion by the translator. If only the `AAAA` record is available, it requests the mapper to assign an IPv4 address corresponding to the IPv6 address, then creates the `A` record for the assigned IPv4 address, and returns the `A` record to the application.

NOTE: This action is similar to that of the DNS ALG (Application Layer Gateway) used in [60]. See also [60].

Address mapper

It maintains an IPv4 address spool. The spool, for example, consists of private addresses [61]. Also, it maintains a table which consists of pairs of an IPv4 address and an IPv6 address.

When the resolver or the translator requests it to assign an IPv4 address corresponding to an IPv6 address, it selects and returns an IPv4 address out of the spool, and 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 only one exception. When initializing the table, it registers a pair of its own IPv4 address and IPv6 address into the table statically.

1.2.3 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, then creates another query to resolve both `A` and `AAAA` records for the host name, and sends it to the server. In this 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 and then the

resolver returns it to the application as is. There is no need for the IP conversion as shown later.

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 4.3 about the influence on other hosts caused by an IPv4 address assigned here.

The application sends an IPv4 packet to "host6."

The IPv4 packet reaches the translator. The translator tries to translate 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. See subsection 2.3.

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 in "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 1.2 illustrates the action described above:

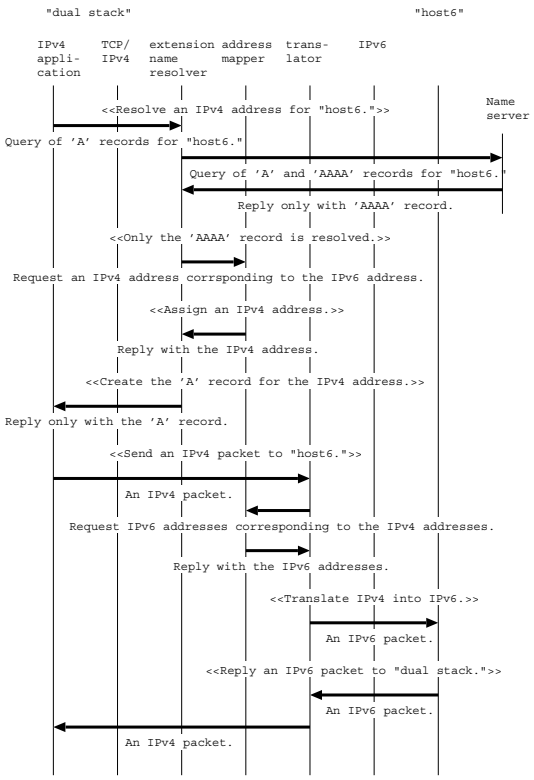


図 1.2　Action of the originator

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 then sends an IPv6 packet to the IPv6 address.

The IPv6 packet reaches the translator in "dual stack."

The translator tries to translate 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. See subsection 2.3.

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 ad-

dress to the translator.

NOTE: See subsection 4.3 about the influence on other hosts caused by an IPv4 address assigned here.

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 1.3 illustrates the action described above:

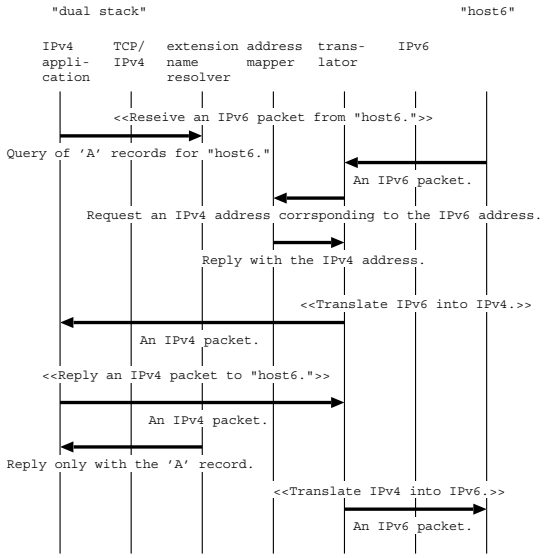


図 1.3 Action of the recipient

1.2.4 Considerations

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

IP conversion

In common with NAT [62], IP conversion needs to translate IP addresses embedded in application layer protocols, which are typically found in FTP [63]. So it is hard to translate all such applications completely.

IPv4 address spool and mapping table

The spool, for example, consists of private addresses [61]. So a large address space can be used for the spool. Nonetheless, IPv4 addresses in the spool will be exhausted and cannot be assigned to IPv6 target hosts, if the host communicates with a great number of 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

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.

1.2.5 Applicability and Limitations

This section considers applicability and limitations of the proposed dual stack hosts.

Applicability

The mechanism can be useful for users in the especially initial stage where some applications not modified into IPv6 remain. And it can also help users who cannot upgrade their certain applications for some reason after all applications have been modified. The reason is that it allows hosts to communicate with IPv6 hosts using existing IPv4 applications, and that they can get connectivity for both IPv4 and IPv6 even if they do not have IPv6 applications as a result.

Note that it can also work in conjunction with a complete IPv6 stack. They can communicate with both IPv4 hosts and IPv6 hosts using IPv4 applications via the mechanism, and can also communicate with IPv6 hosts using IPv6 applications via the complete IPv6 stack.

Limitations

The mechanism is valid only for unicast communication, but invalid for multicast communication. Multicast communication needs another mechanism.

It allows hosts to communicate with IPv6 hosts

using existing IPv4 applications, but this can not be applied to IPv4 applications which use any IPv4 option since it is impossible to translate IPv4 options into IPv6. Similarly it is impossible to translate any IPv6 option headers into IPv4, except for fragment headers and routing headers. So IPv6 inbound communication having the option headers may be rejected.

In common with NAT [62], IP conversion needs to translate IP addresses embedded in application layer protocols, which are typically found in FTP [63]. So it is hard to translate all such applications completely.

It may be impossible that the hosts using the mechanism utilize the security above network layer since the data may carry IP addresses.

Finally it can not combine with secure DNS since the extension name resolver can not handle the protocol.

1.2.6 Security Considerations

This section considers security of the proposed dual stack hosts.

The hosts can utilize the security of all layers like ordinary IPv4 communication when they communicate with IPv4 hosts using IPv4 applications via the mechanism. Likewise they can utilize the security of all layers like ordinary IPv6 communication when they communicate with IPv6 hosts using IPv6 applications via the complete IPv6 stack. However, unfortunately, they can not utilize the security above network layer when they communicate with IPv6 hosts using IPv4 applications via the mechanism. The reason is that when the protocol data with which IP addresses are embedded is encrypted, or when the protocol data is encrypted using IP addresses as keys, it is impossible for the mechanism to translate the IPv4 data into IPv6 and vice versa. Therefore it is highly desirable to upgrade to the applications modified into IPv6 for utilizing the security at communication with IPv6 hosts.

1.3 Possible abuse against IPv6 transition technologies

1.3.1 Abuse of IPv4 compatible address

Problem

To implement automatic tunnelling [64], IPv4 compatible addresses (like ::123.4.5.6) are used. From IPv6 stack point of view, an IPv4 compatible address is considered to be a normal unicast address. If an IPv6 packet has IPv4 compatible addresses in the header, the packet will be encapsulated automatically into an IPv4 packet, with IPv4 address taken from lowermost 4 bytes of the IPv4 compatible addresses. Since there is no good way to check if embedded IPv4 address is sane,

improper IPv4 packet can be generated as a result. Malicious party can abuse it, by injecting IPv6 packets to an IPv4/v6 dual stack node with certain IPv6 source address, to cause transmission of unexpected IPv4 packets. Consider the following scenario:

- You have an IPv6 transport-capable DNS server, running on top of IPv4/v6 dual stack node. The node is on IPv4 subnet 10.1.1.0/24.
- Malicious party transmits an IPv6 UDP packet to port 53 (DNS), with source address ::10.1.1.255. It does not make difference if it is encapsulated into an IPv4 packet, or is transmitted as a native IPv6 packet.
- IPv6 transport-capable DNS server will transmit an IPv6 packet as a reply, copying the original source address into the destination address. Note that the IPv6 DNS server will treat IPv6 compatible address as normal IPv6 unicast address.
- The reply packet will automatically be encapsulated into IPv4 packet, based on RFC1933 automatic tunnelling. As a result, IPv4 packet toward 10.1.1.255 will be

transmitted. This is the subnet broadcast address for your IPv4 subnet, and will (improperly) reach every node on the IPv4 subnet.

Possible solutions

For the following sections, possible soluitions are presented in the order of preference (the author recommends to implement solutions that appear earlier). Note that some of the following are partial solution to the problem. Some of the solutions may overwrap, or be able to coexist, with other solutions.

- Disable automatic tunnelling support.
- Reject IPv6 packets with IPv4 compatible address in IPv6 header fields. Note that we may need to check extension headers as well.
- Perform ingress filter against IPv6 packet and tunnelled IPv6 packet. Ingress filter should let the packets with IPv4 compatible source address through, only if the source address embeds an IPv4 address belongs to the organization. The approach is a partial solution for avoiding possible transmission of malicious packet, from the organization to the outside.
- Whenever possible, check if the addresses on the packet meet the topology you have. For example, if the IPv4 address block for your site is 43.0.0.0/8, and you have a packet from IPv4-wise outside with encapsulated IPv6 source matches ::43.0.0.0/104, it is likely that someone is doing something nasty. This may not be possible to make the filter complete, so consider it as a partial solution.
- Require use of IPv4 IPsec, namely authentication header [65] , for encapsulated packet. Even with IPv4 IPsec, reject the packet if the IPv6 compatible address in the IPv6 header does not embed the IPv4 address in the IPv4 header. We cannot blindly trust the inner IPv6 packet based on the existence of IPv4 IPsec association, since the inner IPv6 packet may be originated by other nodes and forwarded by the authenticated peer. The solution may be impractical, since it only solves very small part of the problem with too many requirements.
- Reject inbound/outgoing IPv6 packets, if it has certain IPv4 compatible address in IPv6 header fields. Note that we may need to check extension headers as well. The author recommends to check any IPv4 compatible address that is mapped from/to IPv4 address not suitable as IPv4 peer. They include 0.0.0.0/8, 127.0.0.0/8, 224.0.0.0/4, 255.255.255.255/32, and subnet broadcast addresses. Since the check can never be perfect (we cannot check for subnet broadcast address in remote site, for example) the direction is not recommend.

1.3.2 Abuse of 6to4 address

6to4 [64] is another proposal for IPv6-over-IPv4 packet encapsulation, and is very similar to RFC1933 automatic tunneling mentioned in the previous section. 6to4 address embeds IPv4 address in the middle (2nd byte to 5th byte). If an IPv6 packet has a 6to4 address as destination address, it will be encapsulated into IPv4 packet with the embedded IPv4 address as IPv4 destination.

IPv6 packets with 6to4 address have the same problems as those with IPv4 compatible address. See the previous section for the details of the problems, and possible solutions.

1.3.3 Abuse of IPv4 mapped address

Problems

IPv6 basic socket API [66] defines the use of IPv4 mapped address with AF_INET6 sockets. IPv4 mapped address is used to handle inbound

IPv4 traffic toward AF_INET6 sockets, and outbound IPv4 traffic from AF_INET6 sockets. Inbound case has higher probability of abuse, while outbound case contributes to the abuse as well. Here we briefly describe the kernel behavior in inbound case. When we have an AF_INET6 socket bound to IPv6 unspecified address (::), IPv4 traffic, as well as IPv6 traffic, will be captured by the socket. The kernel will present the address of the IPv4 peer to the userland program by using IPv4 mapped address. For example, if an IPv4 traffic from 10.1.1.1 is captured by an AF_INET6 socket, the userland program will think that the peer is at ::ffff:10.1.1.1. The userland program can manipulate IPv4 mapped address just like it would do against normal IPv6 unicast address.

We have three problems with the specification. First, IPv4 mapped address support complicates IPv4 access control mechanisms. For example, if you would like to reject accesses from IPv4 clients to a certain transport layer service, it is not enough to reject accesses to AF_INET socket. You will need to check AF_INET6 socket for accesses from IPv4 clients (seen as accesses from IPv4 mapped address) as well.

Secondly, malicious party may be able to use IPv6 packets with IPv4 mapped address, to bypass access control. Consider the following scenario:

- Attacker throws unencapsulated IPv6 packets, with ::ffff:127.0.0.1 as source address.
- The access control code in the server thinks that this is from localhost, and grants accesses.

Lastly, malicious party can make servers generate unexpected IPv4 traffic. This can be accomplished by sending IPv6 packet with IPv4 mapped address as a source (similar to abuse of IPv4 compatible address), or by presenting IPv4 mapped address to servers (like FTP bounce attack [67] from IPv6 to IPv4). The problem is slightly different from the problems with IPv4 compatible addresses and 6to4 addresses, since it does not make use of tunnels. It makes use of certain behavior of userland applications.

Possible solutions

- Reject IPv6 packets, if it has IPv4 mapped address in IPv6 header fields. Note that we may need to check extension headers as well. IPv4 mapped address is internal representation in a node, so doing this will raise no conflicts with existing protocols. We recommend to check the condition in IPv6 input packet processing, and transport layer processing (TCP input and UDP input) to be sure.
- Reject DNS replies, or other host name database replies, which contain IPv4 mapped address. Again, IPv4 mapped address is internal represntation in a node and should never appear on external host name databases.
- Do not route inbound IPv4 traffic to AF_INET6 sockets. When an application would like to accept IPv4 traffic, it should explicitly open AF_INET sockets. You may want to run two applications instead, one for an AF_INET socket, and another for an AF_INET6 socket. Or you may want to make the functionality optional, off by default, and let the userland applications explicitly enable it. This greatly simplifies access control issues. This approach conflicts with what RFC2553 says, however, it should raise no problem with properly- written IPv6 applications. It only affects server programs, ported by assuming the behavior of AF_INET6 listening socket against IPv4 traffic.
- When implementing TCP or UDP stack, explicitly record the wire packet format (IPv4 or IPv6) into connection table. It is unwise

to guess the wire packet format, by existence of IPv6 mapped address in the address pair.

- We should separately fix problems like FTP bounce attack.
- Applications should always check if the connection to AF_INET6 socket is from an IPv4 node (IPv4 mapped address), or IPv6 node. It should then treat the connection as from IPv4 node (not from IPv6 node with special adderss), or reject the connection. This is, however, dangerous to assume that every application implementers are aware of the issue. The solution is not recommended (this is not a solution actually).

1.3.4 Attacks by combining different address formats

Malicious party can use different address formats simultaneously, in a single packet. For example, suppose you have implemented checks for abuse against IPv4 compatible address in automatic tunnel egress module. Bad guys may try to send a native IPv6 packet with 6to4 destination address with IPv4 compatible source address, to bypass security checks against IPv4 compatible address in tunnel decapsulation module. Your implementation will not be able to detect it, since the packet will not visit egress module for automatic tunnels.

Analyze code path with great care, and reject any packets that does not look sane.

1.3.5 Conclusions

IPv6 transition technologies have been proposed, however, some of them looks immune against abuse. The document presented possible ways of abuse, and possible solutions against them. The presented solutions should be reflected to the revision of specifications referenced.

For coming protocols, the author would like to propose a set of guilelines for IPv6 transition technologies:

- Tunnels must explicitly be configured. Manual configuration, or automatic configuration with proper authentication, should be okay.
- Do not embed IPv4 addresses into IPv6 addresses, for tunnels or other cases. It leaves room for abuse, since we cannot practically check if embedded IPv4 address is sane.
- Do not define an IPv6 address format that does not appear on the wire. It complicates access control issues.

The author hopes to see more deployment of native IPv6 networks, where tunnelling technologies become unnecessary.

1.3.6 Security considerations

The document talks about security issues in existing IPv6 related protocol specifications. Possible solutions are provided.

1.4 An IPv6-to-IPv4 transport relay translator

1.4.1 Problem domain

When you deploy an IPv6-only network, you still want to be able to gain access to IPv4-only network resources outside, such as IPv4-only web servers. To solve this problem, many IPv6-to-IPv4 translation technologies are proposed, mainly in the IETF ngtrans working group. The memo describes a translator based on the transport relay technique to solve the same problem.

In this memo, we call this kind of translator “TRT” (transport relay translator). A TRT box locates between IPv6-only hosts and IPv4 hosts and translates TCP,UDP/IPv6 to TCP,UDP/IPv4, vice versa.

Advantages of TRT are as follows:

- TRT is designed to require no extra modification on IPv6-only initiating hosts, nor that on IPv4-only destination hosts. Some other translation mechanisms need extra

modifications on IPv6-only initiating hosts, limiting possibility of deployment.

- The IPv6-to-IPv4 header converters have to take care of path MTU and fragmentation issues. However, TRT is free from this problem.

Disadvantages of TRT are as follows:

- TRT supports connected bidirectional traffic only. The IPv6-to-IPv4 header converters may be able to support other cases, such as unidirectional multicast datagrams.
- TRT needs a stateful TRT box between the communicating peers, just like NAT boxes. While it is possible to place multiple TRT boxes in a site, a transport layer connection will go through particular, a single TRT box. The TRT box thus can be considered a single point of failure, again like NAT boxes. Some other mechanisms, such as SIIT [59], use stateless translator boxes which can avoid a single point of failure.

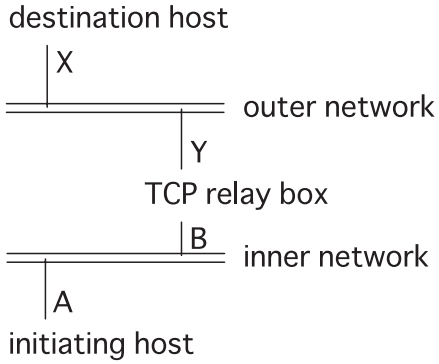
The memo assumes that traffic is initiated by an IPv6-only host destined to an IPv4-only host. The memo can be extended to handle opposite direction, if an apprpriate address mapping mechanism is introduced.

1.4.2 IPv4-to-IPv4 transport relay

To help understanding of the proposal in the next section, here we describe the transport relay in general. The transport relay technique itself is not new, as it has been used in many of firewall-related products.

TCP relay

TCP relay devices have been used in firewall-related products. These products are designed to achieve the follwing goals: (1) disallow forwarding of IP packets across the device, and (2) allow TCP,UDP traffic to go through the device indirectly. For example, consider a network constructed like the following diagram. “TCP relay box” in the diagram does not forward IP packet across the inner network to the outer network, vice versa. It only relays TCP traffic on specific port, from the inner network to the outer network, vice versa. (Note: The diagram has only two subnets, one for inner and one for outer. Actually both side can be more complex, and there can be as many subnets and routers as you wish)



When the initiating host (whose IP address is A) tries to make a TCP connection to the destination host (X), TCP packets are routed toward the TCP relay box based on routing decision. The TCP relay box receives and accepts the packets, even though the TCP relay box does not own the destination IP address (X). The TCP relay box pretends to having IP address X, and establishes TCP connection with the initiating host as X. The TCP relay box then makes a another TCP connection from Y to X, and relays traffic from A to X, and the other way around.

Thus, two TCP connections are established in the picture: from A to B (as X), and from Y to X, like below:

```
TCP/IPv4: the initiating host (A)
          --> the TCP relay box (as X)
    address on IPv4 header: A -> X
TCP/IPv4: the TCP relay box box (Y)
         --> the destination host (X)
    address on IPv4 header: Y -> X
```

The TCP relay box needs to capture some of TCP packets that is not destined to its address. The way to do it is implementation dependent and outside the scope of this memo.

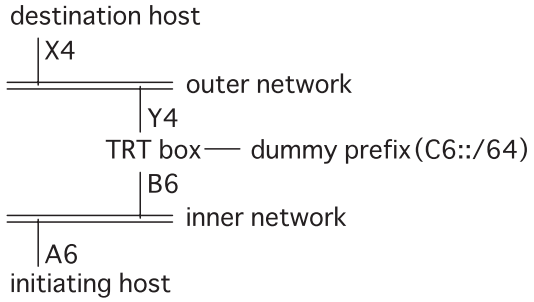
UDP relay

If you can recognize UDP inbound and outbound traffic pair in some way, UDP relay can be implemented in similar manner as TCP relay. An implementation can recognize UDP traffic pair like NAT boxes does, by recording address/port pairs onto an table and managing table entries with timeouts.

1.4.3 IPv6-to-IPv4 transport relay translator

We propose a transport relay translator for IPv6-to-IPv4 protocol translation, TRT. In the following description, TRT for TCP is described. TRT for UDP can be implemented in similar manner.

For address mapping, we will reserve an IPv6 prefix referred to by C6::/64. C6::/64 should be a part of IPv6 unicast address space assigned to the site. Routing information must be configured so that packets to C6::/64 would be routed toward the TRT box. The following diagram shows the network configuration. The subnet marked as "dummy prefix" does not actually exist. Also, now we assume that the initiating host to be IPv6-only, and the destination host to be IPv4-only.



When the initiating host (whose IPv6 address is A6) wishes to make a connection to the destination host (whose IPv4 address is X4), it needs to make an TCP/IPv6 connection toward C6::X4. For example, if C4::/64 equals to fec0:0:0:1::/64, and X4 equals to 10.1.1.1, the destination address to be used is fec0:0:0:1::10.1.1.1. The packet will be routed toward the TRT box, and will be captured by it. The TRT box will accept the TCP/IPv6 connection between A6 and C6::X4, and communicate with the initiating host, using TCP/IPv6. Then, the TRT box will look at the lowermost 32bit of the destination address (IPv6 address C6::X4) to get the real IPv4 destination (IPv4 address X4). It will make an TCP/IPv4 connection from Y4 to X4, and forward traffic across the two TCP connections.

There will be two TCP connections, one is TCP/IPv6 and another is TCP/IPv4, in the picture: from A6 to B6 (as C6::X4), and Y4 to X4, like below:

```
TCP/IPv6: the initiating host (A6)
         --> the TRT box (as C6::X4)
    address on IPv6 header: A6 -> C6::X4
TCP/IPv4: the TRT box (Y4)
         --> the destination host (X4)
    address on IPv4 header: Y4 -> X4
```

1.4.4 Address mapping

As seen in the previous section, an initiating host must use a special form of IPv6 address to connect to an IPv4 destination host. The special form can be resolved from hostname by static address mapping table on the initiating host (like /etc/hosts in UNIX), special DNS

server implementation, or modified DNS resolver implementation on initiating host.

1.4.5 Notes to implementers

TRT for UDP must take care of path MTU issues on the UDP/IPv6 side. This is implementation dependent and outside of the scope of this memo. Simple solution would be to always fragment packets from the TRT box to UDP/IPv6 side to IPv6 minimum MTU (1280 octets), to eliminate the need for path MTU discovery.

Though the TRT box only relays TCP,UDP traffic, it needs to check ICMPv6 packets destined to C6::X4 as well, so that it can recognize path MTU discovery messsages and other notifications between A6 and C6::X4.

When forwarding TCP traffic, a TRT box needs to handle urgent data [111] carefully.

To relay NAT-unfriendly protocols [63] a TRT box may need to modify data content.

Scalability issues must carefully be considered when you deploy TRT boxes to a large IPv6 site. Scalability parameters would be (1) number of connections the operating system kernel can accept, (2) number of connections a userland process can forward (equals to number of filehandles per process), and (3) number of transport relaying processes on a TRT box. Design decision must be made to use proper number of userland processes to support proper number of connections.

To make TRT for TCP more scalable in a large site, it is possible to have multiple TRT boxes in a site. This can be done by taking the following steps: (1) configure multiple TRT boxes, (2) configure different dummy prefix to them, (3) and let the initiating host pick a dummy prefix randomly for load-balancing. (3) can be implemented as follows; If you install special DNS server to the site, you may (3a) configure DNS servers differently to return different dummy prefixes and tell initiating hosts of different DNS servers. Or you can (3b) let DNS server pick a dummy prefix randomly for load-balancing. The load- balancing is possible because you will not be changing destination address (hence the TRT box), once a TCP connection is established.

For address mapping, the authors recommend use of a special DNS server for large-scale installation, and static mapping for small-scale installation. It is not always possible to have special resolver on the initiating host, and assuming it would cause deployment problems.

1.4.6 Security considerations

Malicious party may try to use TRT boxes for anonymizing the source IP address of traffic to IPv4 destinations. TRT boxes should implement

some sort of access control to avoid such improper usage.

A careless TRT implementation may subject to buffer overflow attack, but this kind of issue is implementation dependent and outside the scope of this memo.

A transport relay box hijacks TCP connection between two nodes. This may not be a legitimate behavior for an IP node. The draft does not try to claim it to be legitimate.

1.5 Multiple Destination option on IPv6(MDO6)

1.5.1 Introduction

Current multicast uses the Host Group Model; the destination of a multicast datagram is identified by a Group Multicast Address. Routers that relay datagrams have to maintain routing information for each multicast spanning tree, which causes several scalability problems. [68]

Multicast is useful not only for broadcast applications but also many-to-many applications like a videoconference. Because many-to-many applications need multicast groups much more, the scalability problem described above becomes pretty critical.

Multiple Destination option on IPv6 is a yet another multicast delivery mechanism that depends only on the existing unicast routing environment. The destination of a multicast datagram is specified by a list of unicast addresses instead of a group multicast address. The list is stored in a routing option header with a bitmap that represents the destinations to send on the list. The router looks up the next hop of each unicast address, using their unicast routing table, if their bitmap is on. Then routers copy the datagram and diverge it for the next hops routers sharing the datagrams if any destinations have a common next hop entry.

The routing header has to be evaluated on hop-by-hop basis, The hop-by-hop option header indicates on MDO6 routing header follows. The IPv6 destination field of MDO6 datagrams are filled by one of the unicast addresses of the destinations and the type of the MDO6 Hop-by-hop option has a bit prefix '00' so that routers that cannot recognize MDO6 can treat the MDO6 datagram

as an ordinal IPv6 datagram and forward to one of the destinations. Datagram reachability is preserved because if it passed a preferable router to divide, it can go back at the next MDO6 router the datagram reached.

"Tractable list" is a technology for routing efficiency. A sender is able to sort the destination list so that the on-bits of destination bitmap must appear continuously on whole stems of the spanning tree. In the sparse multicast situation, multicast packets are passed on to almost routers without diverging. In that case, intermediate routers can distinguish the packet that they do not need to diverge only by looking up two addresses located at both ends of the bitmap.

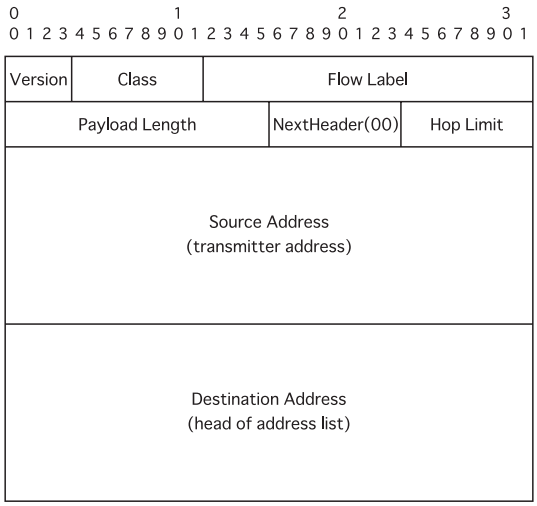
1.5.2 Header Format

MDO6 datagram has 3 extra option headers described below.

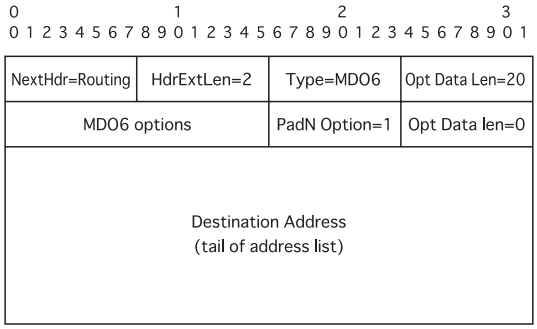
```
IPv6 header
Hop-by-hop options(Type MDO)
Routing(Type MDO explicit destination list)
Destination options(Type MDO)
Other header
Upper protocol header
Payload
```

IPv6 Header

An IPv6 header of MDO6 datagrams has the same format as ordinal IPv6 datagrams. The source address of an IPv6 header is a unicast address of the transmitter. The destination address is a unicast address whose node appears first in the destination bitmap, specified in Section 2.3 and 2.4. Next Header should name "Hop-by-hop option"(00), because MDO6 datagrams must include the MDO6 Hop-by-hop option.



hop-by-hop option header



MDO6 datagrams must have the Hop-by-hop option because they should be checked on every intermediate router on their spanning trees.

The Next Header field and Header Extension Length field must be filled by the type of the next header and length of this hop-by-hop option specified by [125].

The option type number of the MDO6 hop-by-hop option, temporarily in this draft, is XX. (It must be assigned by ICANN in future.) The first 2 bits of XX must be 00 so that an intermediate router that cannot recognize MDO6 skips evaluation of the option and the datagrams are able to pass the router for the IPv6 destination that is one of the destinations on the MDO6 destination list. The third bit of the Hop-by-hop option must be 1 so that the source and destination nodes exclude this options header from targets of the calculation

of Authentication Header, because the destination address field is changed in the multicast delivery path.

MDO6 Options field describes how this datagram should behave. Its value is obtained by applying logical AND to the values below. The meaning of these flags will be explained in section 5.

```
0x01: tractable
0x02: explore branch
0x04: explore branch exhaustively
```

The destination address field has a unicast address whose node appears last in the destination bitmap that makes a pair with the destination field of the IPv6 header.

Options other than the MDO6 option can be packed in the Hop-by-hop option header. Appearance order of the options is not specified, but it is recommended to settle MDO6 options first. It helps in handling hardware evaluation of tractable list handling that is explained in section 2.1.

Routing Header

Complete list of unicast addresses of the destinations is enclosed as a routing optional header. An MDO6 routing header is defined as a variation of an IPv6 routing header specified by RFC1883.

Following accordingly RFC1833 the Next Header and Header Extension Length fields are filled with the type of next header and length of the routing header. The type value in the routing header is YY temporarily.

RFC1833 stipulates that when the 4th octet of the routing header is not 0 and it's type is not recognized by the router, the router must discard the datagram and reply with an ICMP error to the source of the datagram. This enables DoS spoofing attacks, if combined with multicast delivery. So MDO6 strongly recommends that 0 always fills the 4th octet of the routing header. That guarantees that routers that cannot recognize the MDO6 option only discard datagrams without replying with an ICMP error.

The number of destination unicast addresses must be stored in the 5th octet of the routing header. The maximum number of destinations is

1.5.3 This restriction relates to the length limit (8 * 255 octet) of

the routing header itself. The 6th to 8 octet are filled by the padding option.

In the 9th to 24th octet, the destination bitmap is stored, that indicates which unicast destinations in the address list that follows are to be sent to. 1 represents "send-to" and 0 is done. If the number of destinations is less than 126, 0 must fill the following bitmap field.

Destination Option Header

In the IPv6 specification, it has been carefully determined that the ICMP error reply for multi-

cast datagrams must not occur in order to prevent crackers from attacking by smurfing. But MDO6 datagrams are treated as simple unicast datagrams by routers that cannot recognize MDO6. So error reply may occur anyway.

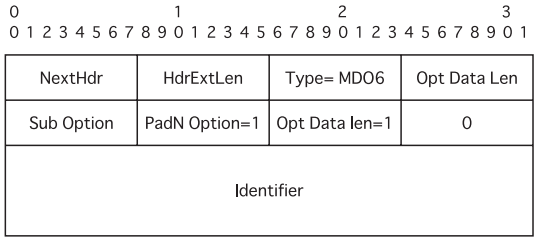
To prevent this behavior, MDO6 datagram must enclose the destination option header. The prefix to 2 bits of the option type is 010 that mean discard the datagram and do not reply with an ICMP error if the node cannot recognize the option. All routers that diverge MDO6 datagrams MUST check whether they have a legal destination option header or not. If they don't have, they MUST just discard the datagrams.

Sub option field is specified additional behavior in the destination node as follows.

```
0: anti-smurfing stopper only.
1: port list
```

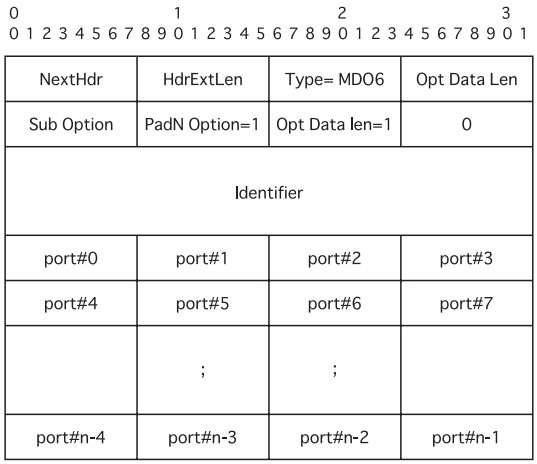
The port list mechanism is described in following section.

The identifier is the last Identifier of the ICMP branch explore when a tractable list is established. If a datagram has no tractable list, the Identifier field must be zero-filled.



1. Port list

In order to deliver the MDO6 datagram for different port of each receiver, transmitter can specify the port list in the destination header as follows.



The numbers of port including the list MUST be equal to the number of destinations specified in the routing header. The length of Option header MUST be 8 octet aligned. The remains field of the port MUST be filled by zero.

The port number field of the UDP header with this option MUST be filled by zero.

1.5.4 Packet delivery

In this section, method of datagram delivery is explained as scenario based behaviors.

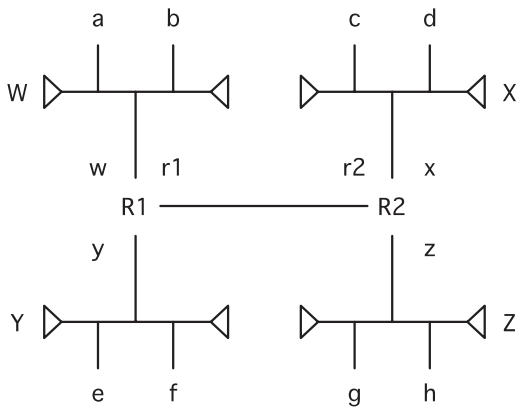
Delivery Scenario with Non-tractable Destination List.

This is a basic scenario. The sender of MDO6 an datagram stores the destination unicast address list in random order and transmits the packet for the node or next hop router.

The router receives the datagram and checks the hop-by-hop option. Because the MDO6 option type represents that it is a non-tractable list, the router starts route evaluation for all unicast addresses on the list of destinations that the destination bitmap is set. The routers choose all unicast addresses that have the same next hop result, set these in destination bitmap to 1, and forward the datagrams to the next hop routers.

When the final receiver captures a datagram, the receiver searches for its unicast address in the destination list, then passes it to the upper protocol layer.

Example:



An example network was built with 8 hosts connected by 4 ethernets(W,X,Y,Z), 1 leased line and 2 routers. Each host has addresses a h. The routers have an interface for leased line(r1,r2) and 2 interfaces for ethernet(w,x,y,z).

The routing table for each host and router is set as below.

[host a,b]

network	gateway	interface
loopback	–	loopback
w	–	ethernet
default	w	ethernet

[host c,d]

network	gateway	interface
loopback	–	loopback
X	–	ethernet
default	x	ethernet

[host e,f]

network	gateway	interface
loopback	–	loopback
Y	–	ethernet
default	y	ethernet

[host g,h]

network	gateway	interface
loopback	–	loopback
Z	–	ethernet
default	z	ethernet

[R1]

network	gateway	interface
loopback	–	loopback
W	–	ethernet(W)
X	r2	r1
Y	–	ethernet(Y)
Z	r2	r1

[R2]

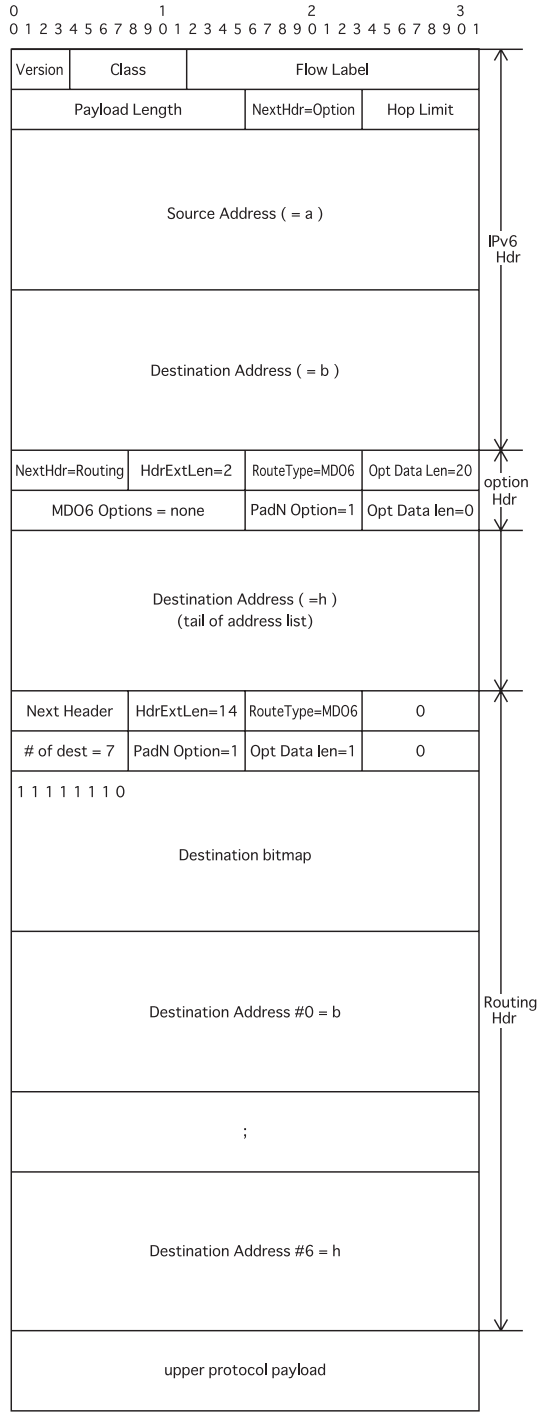
network	gateway	interface
loopback	–	loopback
W	r1	r2
X	–	ethernet(X)
Y	r1	r2
Z	–	ethernet(Z)

A process on host a sends a multicast datagram for destinations b, c, d, e, f, g and h using sendmsg (socket, msg, flags). A parameter msg includes a list of unicast destination addresses in struct iovec style. The protocol stack makes a datagram referencing this iovec as below.

```
IPv6 src   = a
IPv6 dst   = b
IPv6 Opt   = MDO6 followed

MDO6 option = None ( = non tractable )
MDO6 dst    = h

RouteType  = MDO6
# of dest  = 7
dest addr  = [b,c,d,e,f,g,h]
```

```
bitmap      = [1,1,1,1,1,1,1]
```

Here is the detailed datagram format.



Host a looks up the next hop for each address then gets the result below.

```
b:          directly deliverable
            via ethernet W.
c,d,e,f,g,h: relay via router w.
```

For host b, host a rewrites the header as below, then transmit it to b.

```
IPv6 dest   = b
bitmap      = [1,0,0,0,0,0,0]
dest addr   = [b,c,d,e,f,g,h]
```

Host b captures this datagram and checks the MDO0 hop-by-hop option. A destination bitmap in a routing header means only one destination is to be sent to, and it find the destination host is itself. Then it passes the datagram to the upper level protocol stack.

For router w, host a rewrites the bitmap as below.

```
IPv6 dest   = c
bitmap      = [0,1,1,1,1,1,1]
dest addr   = [b,c,d,e,f,g,h]
```

Router R1 captures this datagram and checks the MDO0 hop-by-hop option. It finds the datagram is not for this router and looks up the next hop using the routing table as below.

```
c,d,g,h: to be relay via router r2.
e,f:     directly deliverable
         via ethernet Y.
```

Now, for c,d,g,h, r1 sends the datagram rewriting the bitmap as below.

```
IPv6 dest   = c
bitmap      = [0,1,1,0,0,1,1]
dest addr   = [b,c,d,e,f,g,h]
```

In the same way,

```
IPv6 dest   = e
bitmap      = [0,0,0,1,0,0,0]
dest addr   = [b,c,d,e,f,g,h]
```

for e and

```
IPv6 dest   = f
bitmap      = [0,0,0,0,1,0,0]
dest addr   = [b,c,d,e,f,g,h]
```

and for f.

R2 relays the datagrams as well as R1. Then datagrams can reach every destination.

1.5.5 Impact for Upper Layer Protocol

MDO6 datagrams have option headers that are related to other options and upper layer protocols. In this section, we describe the impact for upper layer protocols.

The fields below for MDO6 options are rewritten as it travels along the delivery path.

- Destination address of IPv6 header.
- Hop-by-hop option header.
- Destinations bitmap in the routing header.

This has an impact on i) checksum calculation in UDP and ICMP headers and ii) IPsec Authentication Header.

i)　Checksum calculation in UDP and ICMP

In both UDP and ICMP, the target of the checksum calculation includes the pseudo header, transport header and payload. Optional headers are not target. In the target, only the destination address of the pseudo header may be rewritten.

UDP and ICMP datagrams on MDO6 must use 0::0(address all zeros) as the destination address of the pseudo header for calculating a checksum.

ii)　IPsec Authentication header

Unlike checksum calculation, optional headers are the target of the calculation of hashed value of the IPsec Authentication header. It must be controlled as to which option should be the target of hash value calculation.

- Hop-by-hop option header and routing header must be rewritten by intermediate routers. Third bit of the MDO6 option type must be 1 so that the hop-by-hop option header is excluded from the calculation of the hash value.
- Destination option header of MDO6 is not rewritten by intermediate routers. So it should be included in target. The third bit of the MDO6 option type must be 0 so that the hop-by-hop option header is included in the calculation of the hash value.

1.5.6 Tractable Order List

As described in section 3, intermediate routers must look up the next hop for all destinations, if addresses are put in random order. It is too expensive. It will not be compatible with the hardware routing facility because the destination list would have variable length.

A tractable ordered list is a list of destination addresses that has been ordered so that the intermediate router can skip looking up the routing entry. An MDO6 application delivers datagrams for a small number of receivers sparsely distributed over the Internet. Most intermediate routers only relay from an interface to an interface without diverging. With a tractable ordered list, such routers can found that they need not to diverge by looking up the routing entry only 2 times.

To make the tractable ordered list, the sender searches the multicast spanning tree and lines up the leaf destinations in depth first search order. With the example described in section 3, the multicast delivery spanning tree is as below.

Then [b,e,f,g,h,d,c] is a result of depth first search, and it is tractable ordered.

A tractable ordered tree has the characteristic that on every stems of the delivery tree, the series

of 1 in the destination bitmap are always continuous. This means that if an intermediate router determines the first and last destinations to be sent have the same next hop, they also can deduce that destinations between the two have the same next hop without looking up the routing entry. In order for intermediate routers to determine easily whether or not they must diverge the MDO6 datagram or not, the first destination address to be sent is stored in the destination address field of the IPv6 header and last destination address is stored in the MDO6 Hop-by-hop option header.

To make the tractable ordered list, three additional types of ICMP datagram(explore branch, branch history and request for re-explore) are introduced. Tractable ordered lists are combined using the procedure outlined below.

1. The sender of MDO6 transmits ICMP_EXPLORE datagrams encapsulated by MDO6 datagrams.
2. Intermediate routers route the ICMP datagrams recording branch histories at diverging points.
3. The receiver sends back the ICMP_EXPLORE datagrams to the transmitter with the branch history.
4. The sender the calculates spanning tree from branch histories.

After the sender has probed the branch environment, the unicast routing may be changed by the alteration of the network topology. That may destroy the tractability of the destination list. Receivers are able to determine some kinds of routing environment changes by observing the TTLs of received datagrams. Receivers notify senders by ICMP_REQ_EXPLORE when they detect the variation of TTLs to recombine tractable ordered list.

Detailed Behavior of MDO6 Datagram with Tractable Ordered List

The example below details how an MDO6 datagram with a tractable ordered list is relayed by intermediate routers. Host a transmits the datagram to destinations c,d,g,h with the MDO6 option header as below.

```
IPv6 src    = a
IPv6 dst    = c
IPv6 Opt    = MDO6 followed

MDO6 option = tractable
MDO6 dst     = h

RouteType   = MDO6
# of dest   = 4

bitmap      = [1,1,1,1]
dest addr   = [c,d,g,h]
```

As explained in section 3, in case the destination list is not tractable, intermediate router R1 must check all destination addresses and determines that all four destinations have same next hop r2.

At this time, MDO6 specifies that the list is tractable ordered. R1 looks up the routing table with IPv6 destination address “c” and with MDO6 dest address “h”. These have the same next hop and it reasons that destination between c and h also has the same next hop without looking it up. Then R1 forwards datagram toward next hop r2.

R2 captures the datagram then looks up with c and h. At this time, it is found that the next hop for c is X and for h is Z. R2 also needs to look up the next hop for d and h. It found X and Z are next hops. R2 also checks the

Toward X, R2 modifies the bitmap of the routing header, destination addresses of IPv6 header and MDO6 hop-by-hop option header and hop limit counter as below, then forwards it.

```
IPv6 src    = a
IPv6 dst    = c
IPv6 Opt    = MDO6 followed

MDO6 option = tractable
```

```
MDO6 dst    = d

RouteType  = MDO6
# of dest  = 4

bitmap     = [1,1,0,0]
dest addr  = [c,d,g,h]
```

The same as in the above, R2 modifies headers as below and forwards toward Z.

```
IPv6 src   = a
IPv6 dst   = g
IPv6 Opt   = MDO6 followed

MDO6 option = tractable
MDO6 dst    = h

RouteType  = MDO6
# of dest  = 4

bitmap     = [0,0,1,1]
dest addr  = [c,d,g,h]
```

Explore Branches

MDO6 system sorts the list of destination addresses into tractable order exploring the topology of the multicast delivery tree. The transmitter sends ICMP datagram assembled as a MDO6 datagram itself. Intermediate routers relays the MDO6 datagram recording the histories of divergence. The exploring datagrams reaches the destination then the receiver sends them back to the transmitter in a unicast ICMP datagram with the branch history. The transmitter collects the histories, analyzes the topology of the multicast spanning tree and sorts the destination list into tractable order.

Two types of ICMP datagrams were needed to explore.

- ICMP MDO6 explore branch
- ICMP MDO6 branch history

Explore branch has 2 modes: effective exploring and exhaustive exploring. Effective exploring means omit exploring sub-trees if the tractable ordered list of a sub-tree is able to be combined trivially.

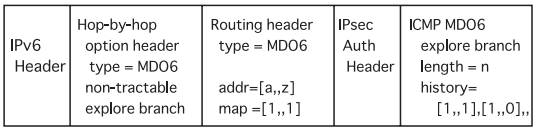
Exploring is needed in the case below.

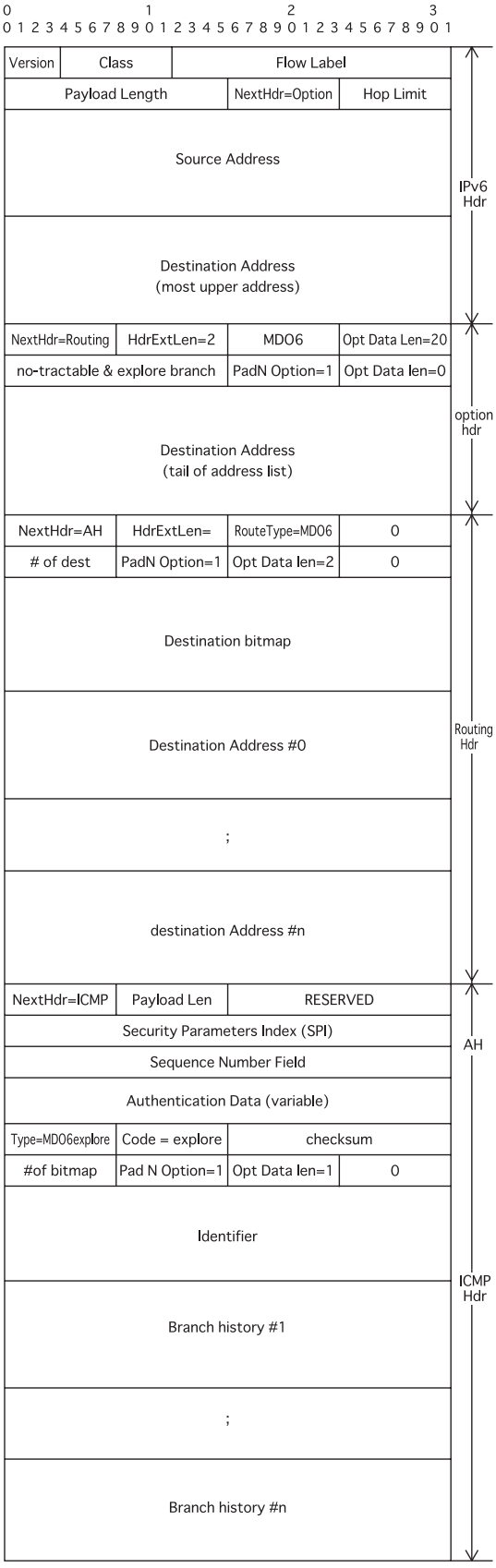
- When a new receiver is subscribed to the list.
- When the routing environment is changed.

It is allowable but not recommended to explore periodically because of the cost of processing on intermediate routers. Instead of periodical exploring, it is recommended for receivers to observe the TTLs of received datagrams. The receiver notifies a transmitter when TTLs change, then the transmitter starts re-exploring.

ICMP explore branches

ICMP explore branches datagrams have the format below.





As well as ordinal MDO6 datagrams, the first address of the destination addresses must be set in the destination address of the IPv6 header of ICMP explore branches. The MDO6 hop-by-hop option must be set no-tractable & explore branches, and tail destination address must be set.

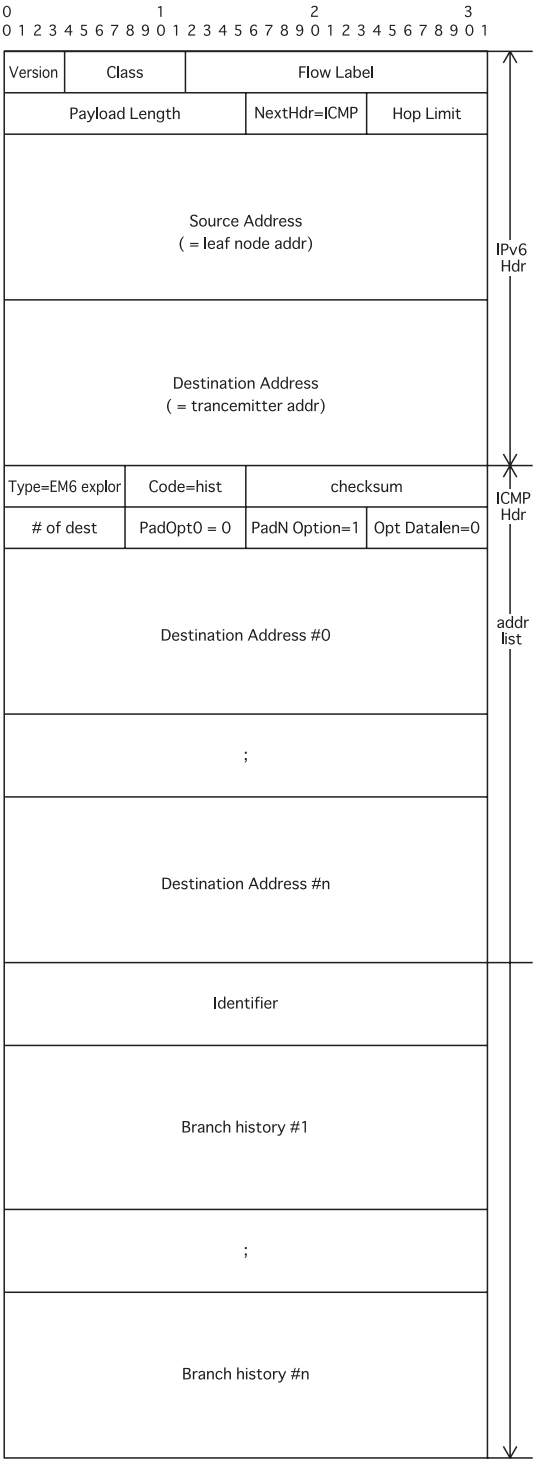
To prevent a source address spoofing attack, the ICMP explore branch must be protected by an IPsec Authentication Header. Because branch history is appended by intermediate routers, the target area or AH hash calculation are from IPv6 header to the identifier field. The transmitter and receivers must exchange Security Association information before exploring branches.

The type field of an ICMP header must be MDO6 explore branch(XXX) and the code field must be explore(YY). Checksum must be a calculated result the same as in ordinal ICMP headers. The number of recorded history must be stored in the # of the bitmap field. The identifier is a unique ID that identifies each explore datagram. It is also embedded in the ICMP branch history datagram so that transmitter can collect result datagrams for the explore request. Branch history bitmaps are a series of branch records.

When intermediate MDO6 routers capture the ICMP explore branch datagrams, they check whether they need to diverge them or not. If they need to diverge them, they append a new destination bitmap to the tail of branch history and increment # of bitmaps. If the exploring mode is exhaustive, they forward the datagram for all next hop routers. If the mode is effective they choose a direction to forward in the way described later.

ICMP branch history

ICMP MDO6 branch history is a type of datagram that contains the result of exploring branches sent from receivers back to the transmitter.



This datagram is delivered by an ordinal unicast IP datagram.

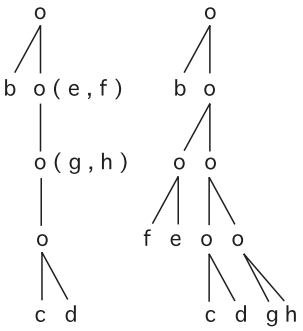
```
IPv6 Header:

  Destination address:
     unicast address of transmitter.

  Source address:
      unicast address
      that send back this history

ICMP header:
  Type: ICMP branch explore(= XX)
  Code: history(= ZZ)

# of destination addresses:
Destination Addresses:
Identifier:
 copy of the ICMP explore branch datagram.

# of bitmap:
  number of branch history records.

Branch history:
  series of history records.
```

Forming a Tractable List from History

As explained in the above section, the transmitter can get branch history records from receivers. Remember the example described in 3.1. A transmitter a wants to sort the destination list [b,c,d,e,f,g,h] then send ICMP MDO6 explore branch datagram. Receivers send back the datagrams below as a result of exploring.

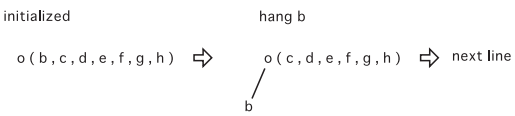
```
   [b,c,d,e,f,g,h]          [b,c,d,e,f,g,h]
   ===============          ===============
b: [1,0,0,0,0,0,0]      c: [0,1,1,1,1,1,1]
                           [0,1,1,0,0,1,1]
                           [0,1,1,0,0,0,0]
                           [0,1,0,0,0,0,0]

   [b,c,d,e,f,g,h]          [b,c,d,e,f,g,h]
   ===============          ===============
d: [0,1,1,1,1,1,1]      e: [0,1,1,1,1,1,1]
   [0,1,1,0,0,1,1]          [0,0,0,1,1,0,0]
   [0,1,1,0,0,0,0]          [0,0,0,1,0,0,0]
   [0,0,1,0,0,0,0]
```

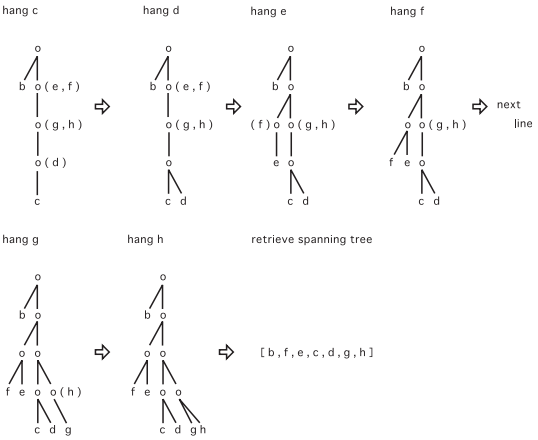
```
   [b,c,d,e,f,g,h]        [b,c,d,e,f,g,h]
   ===============        ===============
f: [0,1,1,1,1,1,1]     g: [0,1,1,1,1,1,1]
   [0,0,0,1,1,0,0]        [0,1,1,0,0,1,1]
   [0,0,0,0,1,0,0]        [0,0,0,0,0,1,1]
                          [0,0,0,0,0,1,0]

   [b,c,d,e,f,g,h]
   ===============
h: [0,1,1,1,1,1,1]
   [0,1,1,0,0,1,1]
   [0,0,0,0,0,1,1]
   [0,0,0,0,0,0,1]

    The transmitter sorts the list by the
    procedure "make_tractable_list()"
    explained below.

  /* branch_tree is temporal data that
     holds such spanning tree structures.
```




```
     o: divergent point
    (e,f) attached by divergent point
    means that the destination that is
    diverged at this point but the
    topology is not yet known below the
    point.

     Right example is final structure
     of analyzing. Left one is
     intermediate one.
   */

   make_tree() {  /* procedure to
   combine spanning tree structure. */
     make root node and attach all nodes
     by the root.
      /* o(b,c,d,e,f,g,h) */

     for ( i = 0; i < # of node ; i++ )
     { /* for all destination nodes */
       check the history result from node_i
       depth = length of history node_i /*
       depth(b) = 1 , depth(h) = 4 */

       hang node_i at a divergent point
       it is attached \\
       with a stem that has new divergent
       points  \\
        (depth - depth(divergent point)).

       foreach( j = i; i < # of nodes; i++ )
       { /* for nodes remains */
         if ( node_j is located on the path
         from root to the node_i ) {
         attach node_j at the divergent
         point that fork with node_i;
         }
     }
   }
   make_regular_list(tree) {
     retrieve tree in depth first order
     and pick nodes up.
   }
   make_tractable_list () {
     make_tree();  /* compose spanning tree */
     make_list();  /* make tractable
     ordered list */
   }
```

With this procedure, spanning tree and tractable list are composed as followed.





Effective Exploring

In the example above, the transmitter can compose a complete spanning tree even if the result of exploring from node h is lost because g returns a branch history that shows that h forks at the last divergent point on the way to g.

Effective exploring is an exploring method cutting unnecessary exploring datagrams at divergent points for nodes that are trivially settled.

To cut off the exploring stems, MDO6 routers group the to-be-sent destinations by the next hop. Next, they count the number of groups that have only one destination and two destinations. 6 cases can be considered.

2 dest　1 dest	no group	1 group	2 or more groups
no group	(i)	(ii)	(iii)
1 or more group	(iii)	(iii)	(iii)

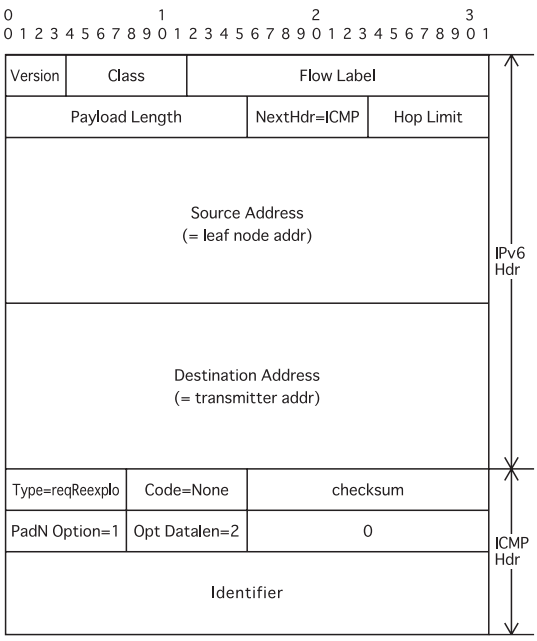
1. All groups have three or more destinations. And they need to be explored for a tractable list. Then transfer ICMP explore history datagram for all next hops.
2. Only one group has one destination. Routers can cut off this group. The transmitter can determine that this node forks at this divergent point by collecting other history branch results.
3. Two or more groups have one destination or one or more groups have two or more destinations. Routers can select and cut off 2 groups that have a destination or 1 group that has 2 destinations, because the transmitter can determine that these 2 nodes fork at this divergent point by collecting other history branch result and it is trivial that 2 variation of sequence of these nodes are also tractable ordered.

The transmitter specifies the modes of exploring by setting MDO6 hop-by-hop options. It is recommended to begin exploring in effective mode. In case exploring fails because of lost datagrams, try to explore exhaustively.

ICMP Re-explore Branch

Because the tractable list that is ordered by exploring the multicast spanning tree is reflect the unicast routing environment, transition of routing environment may break the tractability of the list.

The transmitter can collaborate with a receiver to reform the tractable list. A receiver records the identifier of tractable lists and it's ordinal hop limit count, continuously observes the TTLs of the captured datagrams and notifies the transmitter when it changes. The transmitter can explore the branch again.



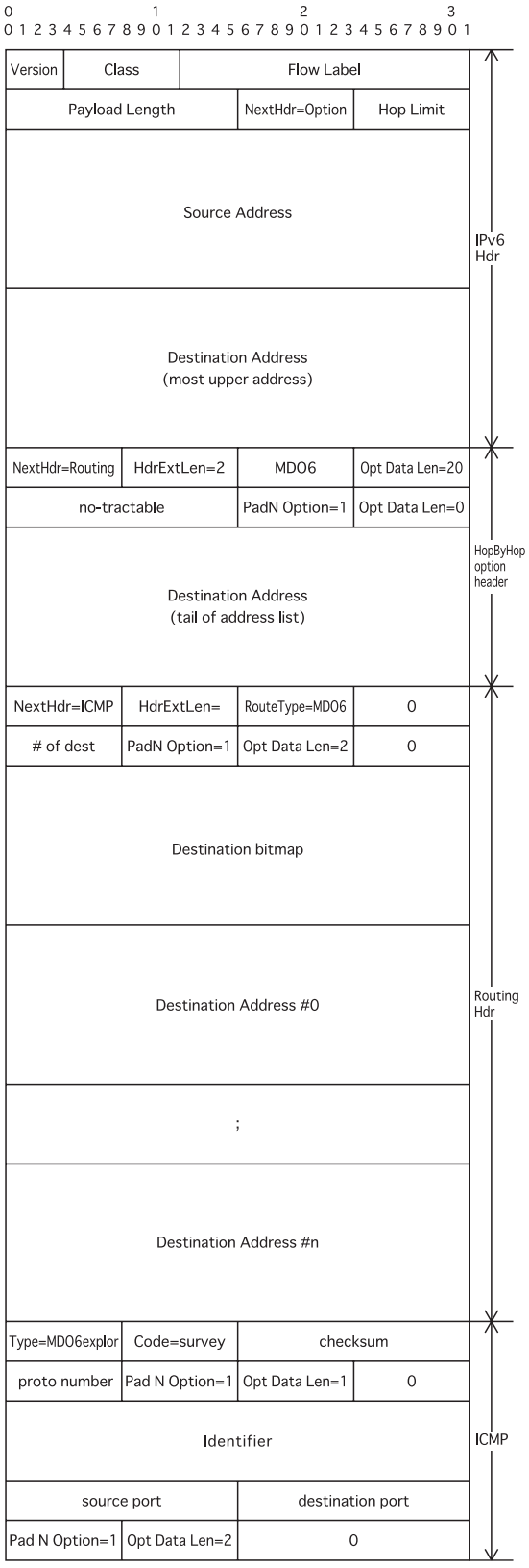
An ICMP type value of MDO6 Request reexplore must be XX, and sub-code must be YY. In

the identifier field the receiver set the last ICMP explore branch receiver captured.

When the transmitter receives the ICMP request re-explore the first time, it starts the exploring procedure. The receiver can notify the transmitter again because ICMP datagrams lost on the way to the transmitter. When the transmitter receives a datagram that has the same identifier again, it must discard it.

ICMP tractability survey

By the hop-limit observation of the ordinal datagrams, there remains possibility that neither transmitter nor receivers cannot detect the tractability in case short cut path is opened. In order to discover the short cut path, transmitters can through the ICMP tractability survey datagram and probe the multicast spanning tree.



This datagram is an ICMP dtagram encapsulated by the MDO6 headers. The type of this

ICMP header is same as other MDO6 ICMP datagrams and the sub code is XY.

In a 5th octet of the header, the type number of the transport protocol that the transmitter use to transfer it ordinal datagrams. In current version of this draft, only UDP(17) is permitted.

Following that, the exploring identifier of the list is stored.

The source and the destination port of the ordinal datagrams must be specified in order to identify the UDP circuit. This field will be varied by the upper transport protocol, in future.

This surveying datagram travels from the transmitter to the receivers as a non-tractable MDO6 datagram while decrementing the hop-limit count. The destination node receive this and search the corresponding UDP circuit. Then it compares the identifier and hop-limit with last one. If the node detect that hop-limit is altered, it requests the transmitters an ICMP re-explore branch in a same way described in Section 5.3 to trigger re-exploring.

1.5.7 Peeling MDO6 option headers

After a MDO6 datagram passed last branching router, it is unnecessary for routers on a remained stem of a spanning tree to check the MDO6 options. In order to avoid the checking, the MDO6 routers may check the number of destinations while diverging the datagram. If the lists has only one destination, they MAY strip the hop-by-hop option and the routing option headers. But routers MUST leave the destination headers because the destination nodes MUST check the destination option to against smurfing, as argued in Section 2.4.

The routers MAY strip the hop-by-hop option and the routing one, while prematurely cloning the datagrams. [68]

1.5.8 Discussion

Security

Many ISP prohibit source routing to prevent packets from passing along the route the ISP cannot control. Using the MDO6 tractable list, a cracker transmits a datagram for the target destination along an illegal relay point while putting target address between relay point address. Routers can discard such illegal datagrams checking the head and tail TLA with other destinations. If head and tail have the same TLA but others are different, the router discards it.

A host that recognizes MDO6 options replies with an ICMP error to the sender. It may cause a spoofing attack. In order to prevent it, MDO6 datagram must include a destination header that causes discarding by a non-MDO6 host. And all MDO6 routers that diverge the datagram MUST check they have legal destination option header or not.

MTU

MDO6 option can have up to 126 destinations. But it shares a 2064 octet length and it is longer than the Ethernet MTU. MDO6 is mainly focused on the usage for small groups of participants. Many multicast applications transmit 1024-length payloads. In this case up to 16 participants can join to the group. (IPv6, UDP, RTP header was under consideration.)

1.6 IPv6 multihoming support at site exit routers

1.6.1 Problem

IPv6 specifications try to decrease the number of backbone routes, to cope with possible memory overflow problem in the backbone routers. To achieve this, the IPv6 addressing architecture [125] only allows the use of aggregatable addresses. Also, IPv6 network administration rules [69] do not allow non-aggregatable routing announcements to the backbone.

In IPv4, a multihomed site uses either of the following technique to achieve better reachability:

- Obtain a portable IPv4 address prefix,

and announce it from multiple upstream providers.

- Obtain single IPv4 address prefix from ISP A, and announce it from multiple upstream providers the site is connected to.

The above two methodologies are not available in IPv6, but on the other hand IPv6 sites and hosts may obtain multiple simultaneous address prefixes to achieve the same result.

The document provides a way to configure site exit routers and ISP routers, so that the site can achieve better reachability from multihoming connectivity, without violating IPv6 rules. The technique uses already-defined routing protocol (BGP or RIPng), and tunnelling of IPv6 packets, and introduces no new protocol standard.

The document is largely based on RFC2260 [70] by Tony Bates.

1.6.2 Goals and non-goals

The goal of this document is to achieve better packet delivery from a site to the outside, or from the outside to the site, even when some of site exit link is down.

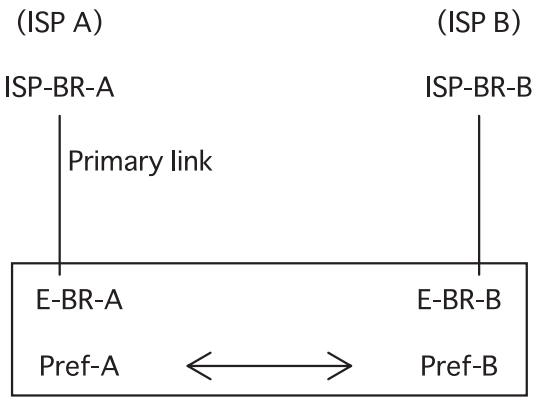
Non goals are:

- Choose the "best" exit link as possible (How do you measure "best" exit link anyway?).
- Achieve load-balancing between multiple exit links.

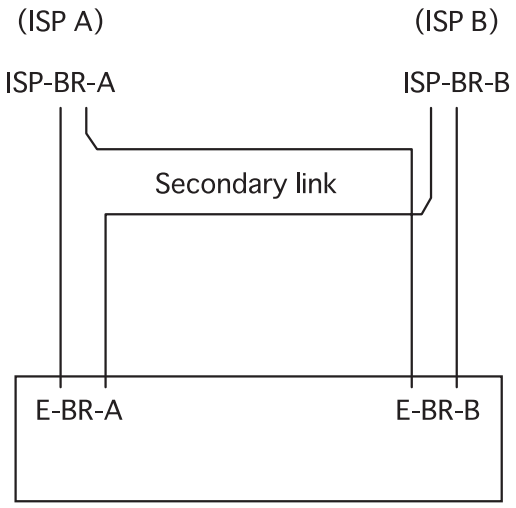
1.6.3 Basic mechanisms

We use technique described in RFC2260 section 5.2 onto our configuration. To summarize, for IPv4-only networks, RFC2260 says that:

- We assume that our site is connected to 2 ISPs, ISP-A and ISP-B.
- We are assigned IP address prefix, Pref-A and Pref-B, from ISP-A and ISP-B respectively. Hosts near ISP-A will get an address from Pref- A, and vice versa.
- In the site, we locally exchanage routes for Pref-A and Pref-B, so that hosts in the site can communicate with each other without using external link.
- ISP-A and our site is connected by "primary link" between ISP router ISP-BR-A and our router E-BR-A. ISP B and our site is connected by primary link between ISP router ISP-BR-B and our router E-BR-B.



- Establish a secondary link, between ISP-BR-A and E-BR-B, and ISP-BR-B and E-BR-A, respectively. Secondary link usually is IP-over-IP tunnel. It is important to have secondary link on top of different medium than primary link, so that one of them survives link failure. For example, secondary link between ISP-BR-A and E-BR-B should go through different medium than primary link between ISP-BR-A and E-BR- A. If secondary link is an IPv4-over-IPv4 tunnel, tunnel endpoint at E-BR-A needs to be an address in Pref-A, not in Pref-B (tunnelled packet needs to travel from ISP-BR-B to E-BR-A, over the primary link between ISP-BR-A and E-BR-A).



- For inbound packets, E-BR-A will advertise (1) Pref-A toward ISP-BR-A with strong preference over primary link, and (2) Pref-B toward ISP- BR-B with weak preference over secondary link. Similarly, E-BR-B will advertise (1) Pref-B toward ISP-BR-B with strong preference over

primary link, and (2) Pref-A toward ISP-BR-A with weak preference over secondary link. Note that we always announce Pref-A to ISP-BR-A, and Pref-B to ISP- BR-B.

- For outbound packets, ISP-BR-A will advertise (1) default route (or specific routes) toward E-BR-A with strong preference over primary link, and (2) default route (or specific routes) toward E-BR-B with weak preference over secondary link. Similarly, ISP-BR-B will advertise (1) default route (or specific routes) toward E-BR-B with strong preference over primary link, and (2) default route (or specific routes) toward E-BR-A with weak preference over secondary link.

Under this configuration, both inbound and outbound packet can survive link failure on either side. Routing information with weak preference will be available as backup, for both inbound and outbound cases.

1.6.4 Extensions for IPv6

RFC2260 is written for IPv4 and BGP. With IPv6 and BGP4+, or IPv6 and RIPng, similar result can be achieved, without violating IPv6 addressing/routing rules.

IPv6 rule conformance

In RFC2260, we announce Pref-A toward ISP-BR-A only, and Pref-B toward ISP-BR-B only. Therefore, there will be no extra routing announcement to the outside of the site. This conforms to the aggregation requirement in IPv6 documents. Also, RFC2260 does not require portable addresses.

Address assignment to the nodes

In IPv4, it is usually assumed that a node will be assigned single IPv4 address. Therefore, RFC2260 assumed that addresses from Pref-A will be assigned to nodes near E-BR-A, and vice versa (second bullet in the previous section).

With IPv6, a node can be assigned multiple IPv6 addresses. So we can assign (1) one address from Pref-A, (2) one address from Pref-B, or (3) two addresses from both address prefixes, to single node in the site.

This will allow more flexibility to nodes in the site. However, this may make source address selection on a node more complex. Source address selection itself is out of scope of the document.

Configuration of links

With IPv6, primary link can be IPv6 native connectivity, RFC1933 [56] IPv6-over-IPv4 configured tunnel, 6to4 [64] IPv6-over-IPv4 encapsulation, or some others.

If tunnelling-based connectivity is used in some of primary links, administrators may want to avoid IPv6-over-IPv6 tunnels for secondary links. For example, if:

- primary links to ISP-A and ISP-B are RFC1933 IPv6-over-IPv4 tunnels, and

- ISP-A, ISP-B and the site have IPv4 con-

nectivity with each other,

it makes no sense to configure a secondary link by IPv6-over-IPv6 tunnel, since it will actually be IPv6-over-IPv6-over-IPv4 tunnel. In this case, IPv6-over-IPv4 tunnel should be used for secondary link. This configuration has a big win as secondary link will be able to have the same path MTU than the primary link.

Using RFC2260 with IPv6 and BGP4+

RFC2260 approach on top of IPv6 will work fine as documented in RFC2260. There will be no extra twists necessary.

Using RFC2260 with IPv6 and RIPng

It is possible to run RFC2260-like configuration with RIPng [Malkin, 1997] , with careful control of metric. Routers in the figure needs to increase RIPng metric on secondary link, to make primary link a preferred path.

If we denote the RIPng metric for route announcement, from router R1 toward router R2, as metric(R1, R2), the invariants that must hold are:

- metric(E-BR-A, ISP-BR-A) < metric(E-BR-B, ISP-BR-A)
- metric(E-BR-B, ISP-BR-B) < metric(E-BR-A, ISP-BR-B)
- metric(ISP-BR-A, E-BR-A) < metric(ISP-BR-A, E-BR-B)
- metric(ISP-BR-B, E-BR-B) < metric(ISP-BR-B, E-BR-A)

Note that smaller metric means stronger route in RIPng.

1.6.5 Issues with ingress filters in ISP

If the upstream ISP imposes ingress filters [71] to outbound traffic, story becomes much more complex. A packet with source address taken from Pref-A must go out from ISP-BR-A. Similarly, a packet with source address taken from Pref-B must go out from ISP-BR-B. Since none of the routers in the site network will route packets based on source address, packets can easily be routed to incorrect border router.

One possible way is to negotiate with both ISPs, to allow both Pref-B and Pref-A to be used as source address. This approach does not work if upstream ISP of ISP-A imposes ingress filtering. Since there will be multiple levels of ISP on top of ISP-A, it will be hard to understand which upstream ISP imposes the filter. In reality, this problem will be very rare, as ingress filter is not suitable for use in large ISPs where smaller ISPs are connected beneath.

Another possibility is to use source-based routing at E-BR-A and E-BR-B. In this case, secondary link needs to be IPv6-over-IPv6 tunnel. When an outbound packet arrives to E-BR-A with source address in Pref-B, E-BR-A will forward it to secondary link (tunnel to ISP-BR-B) based on source- based routing decision. The packet will look like this:

- Outer IPv6 header: source = address of E-BR-A in Pref-A, dest = ISP- BR-B
- Inner IPv6 header: source = address in Pref-B, dest = final dest

Tunneled packet will travel across ISP-BR-A toward ISP-BR-B. The packet can go through ingress filter at ISP-BR-A, since it has outer IPv6 source address in Pref-A. Packet will reach ISP-BR-B and decapsulated before ingress filter is applied. Decapsulated packet can go through ingress filter at ISP-BR-B, since it now has source address in Pref-B (from inner IPv6 header). Notice the following facts when configuring this:

- Not every router implements source-based routing.

- Interaction of normal routing and source-based routing at E-BR-A (and/or E-BR-B) can be vary by router implementations.
- Interaction of tunnel egress and filter rules at ISP-BR-B (and/or ISP-BR-A) can be vary by router implementations and filter configurations.

1.6.6 Observations

We limited the number of ISPs to 2 in the document, but it can easily be extended to the cases where we have 3 or more upstream ISPs.

If you have many upstream providers, you would not make all ISPs backup each other, as it requires $O(N^2)$ tunnels for N ISPs. Rather, it is better to make N/2 pairs of ISPs, and let each pair of ISP backup each other. It is important to pick pairs which are unlikely to be down simultaneously. In this way, number of tunnels will be O(N).

Suppose that the site is very large and it has ISP links in very distant locations, such as in US and in Japan. In such case, it is wiser to use this technique only among ISP links in US, and only among ISP links in Japan. If you use this technique between ISP A in US and ISP B in Japan, the secondary link make packets travel very long path, for example, from host in the site in US, to E-BR-B in Japan, to ISP-BR-B (again in Japan), and then to the final destination in US. This may not make sense for actual use, due to excessive delay.

Similarly, in a large site, addresses must be assigned to end nodes with great care, to minimize delays due to extra path packets may travel. It may be wiser to avoid assigning an address in a prefix assigned from Japanese ISP, to an end node in US.

If one of primary link is down for a long time, administrators may want to control source address selection on end hosts so that secondary link is less likely to be used. This can be achieved by marking unwanted prefix as deprecated. Suppose the primary link toward ISP-A has been down. You will issue router advertisement [Thomson, 1998; Narten, 1998] packets from routers, with preferred lifetime set to 0 in prefix information option for Pref-A. End hosts will consider addresses in Pref-A as deprecated, and will not use any of them as source address for future connections. If an end host in the site makes new connection to outside, the host will use an address in Pref-B as source address, and reply packet to the end host will travel primary link from ISP-BR-B toward E-BR-B.

Some of non-goals (such as “best” exit link selection) can be achieved by combining technique described in this document, with some other techniques. One example of the technique would be the source/destination address selection heuristics on the end nodes.

1.6.7 Security considerations

The configuration described in the document introduces no new security problem.

If primary links toward ISP-A and ISP-B have different security characteristics (like encrypted link and non-encrypted link), administrators needs to be careful setting up secondary links tunneled on them. Packets may travel unwanted path, if secondary links are configured without care.

1.7 Overview of Transition Techniques

1.7.1 Introduction

In the early stage of the migration from IPv4[57] to IPv6[125], it is expected that IPv6 sites will be connected to the IPv4 Internet. On the other hand, in the late stage of the migration, IPv4 sites will be connected to the IPv6 Internet. IPv4 hosts need to be connected to the Internet even 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 both IPv4 hosts and IPv6 hosts in typical environments.
2. A small space of IPv4 address is also assigned to an IPv6 site according to the current severe address assignment policy.
3. An IPv4 site can also obtain a large space of IPv6 address.

In this memo, the word "translator" is used as an intermediate component between an IPv4 host and an IPv6 host to enable direct communication between them, without requiring any modifications to them according to the assumption (1) above.

This memo is organized as follows: Three translation techniques are described in Section 2. Address mapping between IPv4 and IPv6 is discussed in Section 3.

Both SIIT[59] and SOCKS[216] are a kind of translator between IPv4 and IPv6. This memo, however, does not cover such technologies because they require specific modifications to IPv4 and/or IPv6 hosts. BIS[72] is a technology to make an IPv4 host be dual-stack. So, BIS is outside the scope of this memo.

1.7.2 Translation Techniques of IPv4 and IPv6

For translation between IPv4 and IPv6, three technologies are available: header conversion, transport relay, and application level gateway(ALG).

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. (Note that NAT [62] is an IPv4-to-IPv4 header converter.)

The procedure to translate IPv4 packets to IPv6 packets, or vice versa, is defined as a part of SIIT. NATPT[60] (excluding its ALG portion) is an example this kind of translator, which is based on SIIT. (Note that generic concerns about header translation were originally raised in [73].)

Header conversion could be 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[74].

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. Also not all the semantics of ICMP[75] and that of ICMPv6[134] are inter-changeable. However, the latter problem is believed minor in practical cases.

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 with the source host. Then the server also makes one more TCP connection to the real destination. When two connections are established, the server reads data from one of the two connections and writes the data to the other.

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 Level Gateway (ALG)

An ALG for a transaction service is used to hide site information and improve service performance with a cache mechanism. An ALG 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 ALGs for each service must be capable of running over both IPv4 and IPv6

1.7.3 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 translation 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 ALG is free of this problem.

In the case that address mapping is dynamic, it must be implemented in interaction with DNS. 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. However, this violates the assumption (1). So, it is recommended that DNS is used for address mapping even in the static case.

Examples:

An example of static mapping: suppose that an application tries resolving AAAA/A6[76] records against a host name. A DNS server receives this query but it can resolve only `A` record. In this case, the server converts them to AAAA/A6 records embedding them into the pre-configured prefix. Then it returns these records to the application. ([60] also discusses this mechanism.)

An example of dynamic mapping: if a DNS server receives a request to return `A` records for a host name, but only an AAAA/A6 record is resolved, the server picks up an IPv4 address from its address pool then returns it as `A` record.

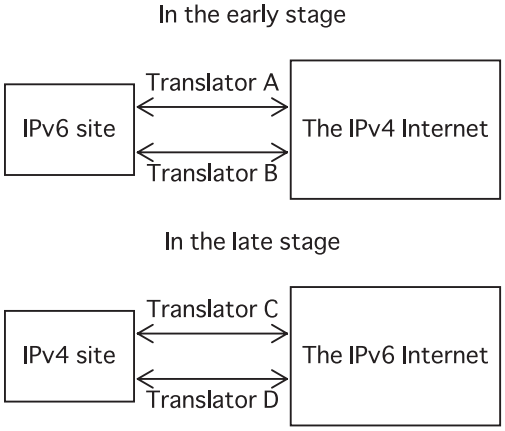
There are two criteria for addresses to be assigned: (1) the assigned addresses must be reachable between a connection initiating host and a translator, and (2) if addresses are assigned dynamically by DNS, it must be ensured that the DNS cache doesn't cause problems for further communications.

If transport relay is used for translation, 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

This memo categorizes IPv4/IPv6 translators from the address mapping point of view. The first picture illustrates the Internet in the early stage of the migration. The second one does that in the late stage.



For simplicity, IPv4/IPv6 translators are categorized into four types. Note that practical translation stories could be combination of these four types.

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

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

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

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

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 Internet Implementation: very hard 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 Internet or to use 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.

Security Consideration

When one or more IPv4/IPv6 translators are used in the intermediate path of an IPv4 host and an IPv6 host, end-to-end authentication mechanisms based on IPv4 and/or IPv6 address (including IPsec[77]) is not available. This problem is well-known in the case of NAT.

1.8 An Extension of Format for IPv6 Scoped Addresses

1.8.1 Introduction

There are several types of scoped addresses defined in the "IPv6 Addressing Architecture" [78]. Since uniqueness of a scoped address is guaranteed only within a corresponding area of the scope, the semantics for a scoped address is ambiguous on a scope boundary. For example, when a user specifies to send a packet from a node to a link-local address of another node, the user must specify the link of the destination as well, if the node is attached to more than one link.

This characteristic of scoped addresses may introduce additional cost to scope-aware applications; a scope-aware application may have to provide a way to specify an instance of a scope for each scoped address (e.g. a specific link for a link-local address) that the application uses. Also, it is hard for a user to "cut and paste" a scoped address due to the ambiguity of its scope.

Applications that are supposed to be used in end hosts like telnet, ftp, and ssh, are not usually aware of scoped addresses, especially of link-local addresses. However, an expert user (e.g. a network administrator) sometimes has to give even

link-local addresses to such applications.

Here is a concrete example. Consider a multi-linked router, called "R1", that has at least two point-to-point interfaces. Each of the interfaces is connected to another router, called "R2" and "R3". Also assume that the point-to-point interfaces are "unnumbered", that is, they have link-local addresses only.

Now suppose that the routing system on R2 hangs up and has to be reinvoked. In this situation, we may not be able to use a global address of R2, because this is a routing trouble and we cannot expect that we have enough routes for global reachability to R2.

Hence we have to login R1 first, and then try to login R2 using link-local addresses. In such a case, we have to give the link-local address of R2 to, for example, telnet. Here we assume the address is fe80::2.

Note that we cannot just type like

```
% telnet fe80::2
```

here, since R1 has more than one interface (i.e. link) and hence the telnet command cannot detect which link it should try to connect.

Although R1 could spray neighbor solicitations for fe80::2 on all links that R1 attaches in order to detect an appropriate link, we cannot completely rely on the result. This is because R3 might also assign fe80::2 to its point-to-point interface and might return a neighbor advertisement faster than R2. There is currently no mechanism to (automatically) resolve such conflict. Even if we had one, the administrator of R3 might not accept to change the link-local address especially when R3 belongs to a different organization from R1's.

This document defines an extension of the format for scoped addresses in order to overcome this inconvenience. Using the extended format with some appropriate library routines will make scope-aware applications simpler.

1.8.2 Assumptions and Definitions

In this document we adopt the same assumption of characteristics of scopes as described in the scoped routing document [79].

We use the term "scope zone" to represent a particular instance of a scope in this document. Note, however, that the terminology for such a notion is to be defined in a separate document.

1.8.3 Proposal

The proposed format for scoped addresses is as follows:

```
<scoped_address>%<scope_id>
```

where <scoped_address> is a literal IPv6 address, <scope_id> is a string to identify the scope of the address, and '%' is a delimiter character to distinguish between <scoped_address> and <scope_id>.

The following subsections describe detail definitions and concrete examples of the format.

Scoped Addresses

The proposed format is applied to all kinds of unicast and multicast scoped addresses, that is, all non-global unicast and multicast addresses.

The format should not be used for global addresses. However, an implementation which handles addresses (e.g. name to address mapping functions) MAY allow users to use such a notation.

Scope Identifiers

An implementation SHOULD support at least numerical identifiers as <scope_id>, which are non-negative decimal numbers. Positive identifiers MUST uniquely specifies a single instance of scope for a given scoped address. An implementation MAY use zero to have a special meaning, for example, a meaning that no instance of scope is specified.

An implementation MAY support other kinds of strings as <scope_id> unless the strings conflict with the delimiter character. The precise semantics of such additional strings is implementation dependent.

One possible candidate of such strings would be interface names, since interfaces uniquely disambiguate any type of scopes [79]. In particular, if an implementation can assume that there is a one-to-one mapping between links and interfaces (and the assumption is usually reasonable,) using interface names as link identifiers would be natural.

An implementation could also use interface names as <scope_id> for larger scopes than links, but there might be some confusion in such use. For example, when more than one interface belongs to a same site, a user would be confused about which interface should be

used. Also, a mapping function from an address to a name would encounter a same kind of problem when it prints a scoped address with an interface name as a scope identifier. This document does not specify how these cases should be treated and leaves it implementation dependent.

It cannot be assumed that a same identifier is common to all nodes in a scope zone. Hence the proposed format MUST be used only within a node and MUST NOT be sent on a wire.

Examples

Here are examples. The following addresses

```
fe80::1234 (whose link identifier is 1)
fec0::5678 (whose site identifier is 2)
ff02::9abc (whose link identifier is 5)
ff08::def0 (whose organization identifier is 10)
```

would be represented as follows:

```
fe80::1234%1
fec0::5678%2
ff02::9abc%5
ff08::def0%10
```

If we use interface names as <scope_id>, the followings could also be represented as follows:

```
fe80::1234%ne0
fec0::5678%ether2
ff02::9abc%pvc1.3
ff08::def0%interface10
```

where the interface “ne0” belongs to link 1, “ether2” belongs to site 2, and so on.

Omitting Scope Identifiers

This document does not intend to invalidate the original format for scoped addresses, that is, the format without the scope identifier portion. An implementation SHOULD rather provide a user with a “default” instance of each scope and allow the user to omit scope identifiers.

Also, when an implementation can assume that there is no ambiguity of any type of scopes on a node, it MAY even omit the whole functionality to handle the proposed format. An end host with a single interface would be an example of such a case.

1.8.4 Combinations of Delimiter Characters

There are other kinds of delimiter characters defined for IPv6 addresses. In this section, we describe how they should be combined

with the proposed format for scoped addresses.

The IPv6 addressing architecture [78] also defines the syntax of IPv6 prefixes. If the address portion of a prefix is scoped one and the scope should be disambiguated, the address portion SHOULD be in the proposed format. For example, the prefix fec0:0:0:1::/64 on a site whose identifier is 2 should be represented as follows:

```
fec0:0:0:1::%2/64
```

There is the preferred format for literal IPv6 addresses in URL’s [80]. When a user types the preferred format for an IPv6 scoped address and the scope should be explicitly specified, the address part in brackets SHOULD be in the proposed format. Thus, for instance, the user should type as follows:

```
http://[fec0:0:0:2::1234%10]:80/index.html
```

1.8.5 Related Issues

In this document, it is assumed that an identifier of a scope is not necessarily common in a

表 1.1　KAME のマージ状況

	IPv4-IPsec	IPv6	IPv6-IPsec	KAME パッチ
BSD/OS 3	なし	なし	なし	あり
BSD/OS 4.1	NRL	NRL	NRL	あり
FreeBSD 2.2.8	なし	なし	なし	あり
FreeBSD 3.4	なし	なし	なし	あり
FreeBSD 4.0	KAME	KAME	KAME	あり
FreeBSD-current	KAME	KAME	KAME	なし
NetBSD 1.4.2	なし	なし	なし	あり
NetBSD-current	KAME	KAME	KAME	なし
OpenBSD 2.6	オリジナル	なし	なし	あり
OpenBSD-current	オリジナル	KAME	オリジナル	なし

scope zone. However, it would be useful if a common notation is introduced (e.g. an organization name for a site). In such a case, the proposed format could be commonly used to designate a single interface (or a set of interfaces for a multicast address) in a scope zone.

When the network configuration of a node changes, the change may affect <scope_id>. Suppose that the case where numerical identifiers are sequentially used as <scope_id>. When a network interface card is newly inserted in the node, some identifiers may have to be renumbered accordingly. This would be inconvenient, especially when addresses with the numerical identifiers are stored in non-volatile storage and reused after rebooting.

1.8.6 Security Considerations

The use of this approach to represent IPv6 scoped addresses does not introduce any known new security concerns, since the use is restricted within a single node.

1.9 KAME のマージ状況

最後に KAME ソフトウェアのマージ状況についてまとめる。

今までバラバラだった各 BSD の IPv6、IPv4-IPsec、IPv6-IPsec コードは、KAME パッケージを採用した。現状を表 1.1 にまとめる。2000 年 3 月現在 KAME が正式に組み込まれているのは、{FreeBSD,NetBSD,OpenBSD}-current である。その他の BSD の各バージョンには、パッチ形式として KAME パッケージを提供している。なお、BSD/OS も KAME に移行する作業が進められている。