Network Working Group
Request for Comments: 3338
Category: Experimental

S. Lee
M-K. Shin
Y-J. Kim
ETRI
E. Nordmark
A. Durand
Sun Microsystems

October 2002

Dual Stack Hosts Using "Bump-in-the-API" (BIA)

Status of this Memo

Copyright Notice

Abstract

This document specifies a mechanism of dual stack hosts using a technique called "Bump-in-the-API"(BIA) which allows for the hosts to communicate with other IPv6 hosts using existing IPv4 applications. The goal of this mechanism is the same as that of the Bump-in-the-stack mechanism, but this mechanism provides the translation method between the IPv4 APIs and IPv6 APIs. Thus, the goal is simply achieved without IP header translation.

Table of Contents:

1. Introduction 2
2. Applicability and Disclaimer 3
2.1 Applicability 3
2.2 Disclaimer 4
3. Dual Stack Host Architecture Using BIA 4
3.1 Function Mapper 4
3.2 Name Resolver 5
3.3 Address Mapper 5
4. Behavior Example 6
4.1 Originator Behavior 6
4.2 Recipient Behavior 8
5. Considerations 10
5.1 Socket API Conversion 10
5.2 ICMP Messages Handling 10
5.3 IPv4 Address Pool and Mapping Table 10
5.4 Internally Assigned IPv4 Addresses 10
5.5 Mismatch Between DNS Result and Peer Application Version 11
5.6 Implementation Issues 11
6. Limitations 12
7. Security Considerations 12
8. Acknowledgments 12
9. References 12
Appendix: API list intercepted by BIA 14
Authors Addresses 16
Full Copyright Statement 17

1. Introduction

RFC2767 [BIS] specifies a host translation mechanism using a technique called "Bump-in-the-Stack". It translates IPv4 into IPv6, and vice versa using the IP conversion mechanism defined in [SIIT]. BIS allows hosts to communicate with other IPv6 hosts using existing IPv4 applications. However, this approach is to use an API translator which is inserted between the TCP/IP module and network card driver, so that it has the same limitations as the [SIIT] based IP header translation methods. In addition, its implementation is dependent upon the network interface driver.

This document specifies a new mechanism of dual stack hosts called Bump-in-the-API(BIA) technique. The BIA technique inserts an API translator between the socket API module and the TCP/IP module in the dual stack hosts, so that it translates the IPv4 socket API function into IPv6 socket API function and vice versa. With this mechanism, the translation can be simplified without IP header translation.

Using BIA, the dual stack host assumes that there exists both TCP(UDP)/IPv4 and TCP(UDP)/IPv6 stacks on the local node.

When IPv4 applications on the dual stack communicate with other IPv6 hosts, the API translator detects the socket API functions from IPv4 applications and invokes the IPv6 socket API functions to communicate with the IPv6 hosts, and vice versa. In order to support communication between IPv4 applications and the target IPv6 hosts, pooled IPv4 addresses will be assigned through the name resolver in the API translator.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119].

This document uses terms defined in [IPv6],[TRANS-MECH] and [BIS].

2. Applicability and Disclaimer

2.1 Applicability

The main purposes of BIA are the same as BIS [BIS]. It makes IPv4 applications communicate with IPv6 hosts without any modification of those IPv4 applications. However, while BIS is for systems with no IPv6 stack, BIA is for systems with an IPv6 stack, but on which some applications are not yet available on IPv6 and source code is not available preventing the application from being ported. It's good for early adopters who do not have all applications handy, but not for mainstream production usage.

There is an issue about a client node running BIA trying to contact a dual stack node on a port number that is only associated with an IPv4 application (see section 5.5). There are 2 approaches.

- The client application SHOULD cycle through all the addresses and end up trying the IPv4 one.

- BIA SHOULD do the work.

It is not clear at this time which behavior is desirable (it may very well be application dependent), so we need to get feedback from experimentation.

2.2 Disclaimer

BIA SHOULD NOT be used for an IPv4 application for which source code is available. We strongly recommend that application programmers SHOULD NOT use this mechanism when application source code is available. As well, it SHOULD NOT be used as an excuse not to port software or delay porting.

3. Dual Stack Host Architecture Using BIA

Figure 1 shows the architecture of the host in which BIA is installed.

```
            +----------------------------------------------+
            | +------------------------------------------+ |
            | |                                          | |
            | |             IPv4 applications            | |
            | |                                          | |
            | +------------------------------------------+ |
            | +------------------------------------------+ |
            | |           Socket API (IPv4, IPv6)        | |
            | +------------------------------------------+ |
            | +-[ API translator]------------------------+ |
            | | +-----------+ +---------+ +------------+ | |
            | | | Name      | | Address | | Function   | | |
            | | | Resolver  | | Mapper  | | Mapper     | | |
            | | +-----------+ +---------+ +------------+ | |
            | +------------------------------------------+ |
            | +--------------------+ +-------------------+ |
            | |                    | |                   | |
            | |    TCP(UDP)/IPv4   | |   TCP(UDP)/IPv6   | |
            | |                    | |                   | |
            | +--------------------+ +-------------------+ |
            +----------------------------------------------+
```

Figure 1 Architecture of the dual stack host using BIA

Dual stack hosts defined in RFC2893 [TRANS-MECH] need applications, TCP/IP modules and addresses for both IPv4 and IPv6. The proposed hosts in this document have an API translator to communicate with other IPv6 hosts using existing IPv4 applications. The API translator consists of 3 modules, a name resolver, an address mapper and a function mapper.

3.1 Function Mapper

It translates an IPv4 socket API function into an IPv6 socket API function, and vice versa.

When detecting the IPv4 socket API functions from IPv4 applications, it intercepts the function call and invokes new IPv6 socket API functions which correspond to the IPv4 socket API functions. Those IPv6 API functions are used to communicate with the target IPv6 hosts. When detecting the IPv6 socket API functions from the data received from the IPv6 hosts, it works symmetrically in relation to the previous case.

3.2 Name Resolver

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

When an IPv4 application tries to resolve names via the resolver library (e.g. gethostbyname()), BIA intercept the function call and instead call the IPv6 equivalent functions (e.g. getnameinfo()) that will resolve both A and AAAA records.

If the AAAA record is available, it requests the address 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.

3.3 Address Mapper

It internally maintains a table of the pairs of an IPv4 address and an IPv6 address. The IPv4 addresses are assigned from an IPv4 address pool. It uses the unassigned IPv4 addresses (e.g., 0.0.0.1 ~ 0.0.0.255).

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

(1) When the name 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 function mapper gets a socket API function call from the data received and there is not a mapping entry for the IPv6 source address.

NOTE: This is the same as that of the Address Mapper in [BIS].

4. Behavior Examples

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

In this section, the meanings of arrows are as follows:

```
---> A DNS message for name resolving created by the applications
     and the name resolver in the API translator.
+++> An IPv4 address request to and reply from the address mapper
     for the name resolver and the function mapper.
===> Data flow by socket API functions created by the
     applications and the function mapper in the API translator.
```

4.1 Originator Behavior

This sub-section describes the behavior when the "dual stack" sends data to "host6".

When an IPv4 application sends a DNS query to its name server, the name resolver intercepts the query and then creates a new query to resolve both A and AAAA records. When only the AAAA record is resolved, the name resolver requests the address mapper to assign an IPv4 address corresponding to the IPv6 address.

The name resolver creates an A record for the assigned IPv4 address and returns it to the IPv4 applications.

In order for the IPv4 application to send IPv4 packets to host6, it calls the IPv4 socket API function.

The function mapper detects the socket API function from the application. If the result is from IPv6 applications, it skips the translation. In the case of IPv4 applications, it requires an IPv6 address to invoke the IPv6 socket API function, thus the function mapper requests an IPv6 address to the address mapper. The address mapper selects an IPv4 address from the table and returns the destination IPv6 address. Using this IPv6 address, the function mapper invokes an IPv6 socket API function corresponding to the IPv4 socket API function.

When the function mapper receives an IPv6 function call,it requests the IPv4 address to the address mapper in order to translate the IPv6 socket API function into an IPv4 socket API function. Then, the function mapper invokes the socket API function for the IPv4 applications.

Figure 2 illustrates the behavior described above:

```
"dual stack"                                                "host6"
IPv4    Socket |     [ API Translator ]    | TCP(UDP)/IP          Name
appli-  API    |Name      Address  Function| (v6/v4)             Server
cation         |Resolver  Mapper   Mapper  |
 |        |        |        |        |         |              |       |
<<Resolve an IPv4 address for "host6".>>       |              |       |
 |        |        |        |        |         |              |       |
 |--------|------->|  Query of 'A' records for host6.         |       |
 |        |        |        |        |         |              |       |
 |        |        |--------|--------|---------|--------------|------>|
 |        |        |  Query of 'A' records and 'AAAA' for host6       |
 |        |        |        |        |         |              |       |
 |        |        |<-------|--------|---------|--------------|-------|
 |        |        |  Reply with the 'AAAA' record.           |       |
 |        |        |        |        |         |              |
 |        |        |<<The 'AAAA' record is resolved.>>        |
 |        |        |        |        |         |              |
 |        |        |+++++++>|  Request one IPv4 address       |
 |        |        |        |  corresponding to the IPv6 address.
 |        |        |        |        |         |              |
 |        |        |        |<<Assign one IPv4 address.>>     |
 |        |        |        |        |         |              |
 |        |        |<+++++++|  Reply with the IPv4 address.   |
 |        |        |        |        |         |              |
 |        |        |<<Create 'A' record for the IPv4 address.>>
 |        |        |        |        |         |              |
 |<-------|--------| Reply with the 'A' record.|              |
 |        |        |        |        |         |              |
```

Figure 2 Behavior of the originator (1/2)

```
"dual stack"                                               "host6"
IPv4    Socket |     [ API Translator ]    | TCP(UDP)/IP
appli-  API    |Name      Address  Function| (v6/v4)
cation         |Resolver  Mapper   Mapper  |
 |        |        |        |        |         |              |
<<Call IPv4 Socket API function >>   |         |              |
 |        |        |        |        |         |              |
 |========|========|========|=======>|An IPv4 Socket API function Call
 |        |        |        |        |         |              |
 |        |        |        |<+++++++|  Request IPv6 addresses|
 |        |        |        |        |  corresponding to the  |
 |        |        |        |        |  IPv4 addresses.       |
 |        |        |        |        |         |              |
 |        |        |        |+++++++>| Reply with the IPv6 addresses.
 |        |        |        |        |         |              |
 |        |        |        |        |<<Translate IPv4 into IPv6.>>
 |        |        |        |        |         |              |
 |  An IPv6 Socket API function call.|=========|=============>|
 |        |        |        |        |         |              |
 |        |        |        |        |<<Reply an IPv6 data    |
 |        |        |        |        |  to dual stack.>>      |
 |        |        |        |        |         |              |
 |  An IPv6 Socket API function call.|<========|==============|
 |        |        |        |        |         |              |
 |        |        |        |        |<<Translate IPv6 into IPv4.>>
 |        |        |        |        |         |              |
 |        |        |        |<+++++++|  Request IPv4 addresses|
 |        |        |        |        |  corresponding to the  |
 |        |        |        |        |  IPv6 addresses.       |
 |        |        |        |        |         |              |
 |        |        |        |+++++++>| Reply with the IPv4 addresses.
 |        |        |        |        |         |              |
 |<=======|========|========|========|  An IPv4 Socket function call.
 |        |        |        |        |         |              |
```

Figure 2 Behavior of the originator (2/2)

4.2 Recipient Behavior

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

"host6" resolves the address of "dual stack" with 'AAAA' records through its name server, and then sends an IPv6 packet to the "dual stack".

The IPv6 packet reaches the "dual stack" and the function mapper detects it.

The function mapper requests the IPv4 address to the address mapper in order to invoke the IPv4 socket API function to communicate with the IPv4 application. Then the function mapper invokes the corresponding IPv4 socket API function for the IPv4 applications corresponding to the IPv6 functions.

Figure 3 illustrates the behavior described above:

```
"dual stack"                                                "host6"
IPv4    Socket |     [ API Translator ]    | TCP(UDP)/IP
appli-  API    |Name      Address  Function| (v6/v4)
cation         |Resolver  Mapper   Mapper  |
 |        |        |        |        |         |              |
<<Receive data from "host6".>>       |         |              |
 |        |        |        |        |         |              |
 |      An IPv6 Socket function call.|<========|==============|
 |        |        |        |        |         |              |
 |        |        |        |<+++++++|  Request IPv4 addresses|
 |        |        |        |        |  corresponding to the IPv6
 |        |        |        |        |  addresses.            |
 |        |        |        |        |         |              |
 |        |        |        |+++++++>| Reply with the IPv4 addresses.
 |        |        |        |        |         |              |
 |        |        |        |        |<<Translate IPv6 into IPv4.>>
 |        |        |        |        |         |              |
 |<=======|========|========|========|  An IPv4 function call |
 |        |        |        |        |         |              |
<<Reply an IPv4 data to "host6".>>   |         |              |
 |        |        |        |        |         |              |
 |========|========|========|=======>|  An IPv4 function call |
 |        |        |        |        |         |              |
 |        |        |        |        |<<Translate IPv4 into IPv6.>>
 |        |        |        |        |         |              |
 |        |        |        |<+++++++|  Request IPv6 addresses|
 |        |        |        |        |  corresponding to the IPv4
 |        |        |        |        |  addresses.            |
 |        |        |        |        |         |              |
 |        |        |        |+++++++>| Reply with the IPv6 addresses.
 |        |        |        |        |         |              |
 |      An IPv6 Socket function call.|=========|=============>|
 |        |        |        |        |         |              |
```

Figure 3 Behavior of Receiving data from IPv6 host

5. Considerations

5.1 Socket API Conversion

IPv4 socket API functions are translated into semantically the same IPv6 socket API functions and vice versa. See Appendix A for the API list intercepted by BIA. IP addresses embedded in application layer protocols (e.g., FTP) can be translated in API functions. Its implementation depends on operating systems.

NOTE: Basically, IPv4 socket API functions are not fully compatible with IPv6 since the IPv6 has new advanced features.

5.2 ICMP Message Handling

When an application needs ICMP messages values (e.g., Type, Code, etc.) sent from a network layer, ICMPv4 message values MAY be translated into ICMPv6 message values based on [SIIT], and vice versa. It can be implemented using raw socket.

5.3 IPv4 Address Pool and Mapping Table

The address pool consists of the unassigned IPv4 addresses. This pool can be implemented at different granularity in the node e.g., a single pool per node, or at some finer granularity such as per user or per process. However, if a number of IPv4 applications communicate with IPv6 hosts, the available address spaces will be exhausted. As a result, it will be impossible for IPv4 applications to communicate with IPv6 nodes. It requires smart management techniques for address pool. For example, it is desirable for the mapper to free the oldest entry and reuse the IPv4 address for creating a new entry. This issues is the same as [BIS]. In case of a per-node address mapping table, it MAY cause a larger risk of running out of address.

5.4 Internally Assigned IPv4 Addresses

The IPv4 addresses, which are internally assigned to IPv6 target hosts out of the pool, are the unassigned IPv4 addresses (e.g., 0.0.0.1 ~ 0.0.0.255). There is no potential collision with another use of the private address space when the IPv4 address flows out from the host.

5.5 Mismatch between DNS result(AAAA) and Peer Application Version(v4)

If a server application you are using does not support IPv6 yet, but runs on a machine that supports other IPv6 services and this is listed with a AAAA record in the DNS, a client IPv4 application using BIA might fail to connect to the server application, because there is a mismatch between DNS query result (i.e., AAAA) and a server application version(i.e., IPv4). A solution is to try all the addresses listed in the DNS and just not fail after the first attempt. We have two approaches: the client application itself SHOULD cycle through all the addresses and end up trying the IPv4 one. Or it SHOULD be done by some extensions of name resolver and API translator in BIA. For this, BIA SHOULD do iterated jobs for finding the working address used by the other application out of addresses returned by the extended name resolver. It may very well be application dependent. Note that BIA might be able to do the iteraction over all addresses for TCP sockets, since BIA can observe when the connect call fails. But for UDP sockets it is hard if not impossible for BIA to know which address worked, hence the application must do the iteraction over all addresses until it finds a working address.

Another way to avoid this type of problems is to make BIA only come into effect when no A records exist for the peer. Thus traffic from an application using BIA on a dual-stack host to a dual-stack host would use IPv4.

5.6 Implementation Issues

Some operating systems support the preload library functions, so it is easy to implement the API translator by using it. For example, the user can replace all existing socket API functions with user-defined socket API functions which translate the socket API function. In this case, every IPv4 application has its own translation library using a preloaded library which will be bound into the application before executing it dynamically.

Some other operating systems support the user-defined layered protocol allowing a user to develop some additional protocols and put them in the existing protocol stack. In this case, the API translator can be implemented as a layered protocol module.

In the above two approaches, it is assumed that there exists both TCP(UDP)/IPv4 and TCP(UDP)/IPv6 stacks and there is no need to modify or to add a new TCP-UDP/IPv6 stack.

6. Limitations

In common with [NAT-PT], BIA needs to translate IP addresses embedded in application layer protocols, e.g., FTP. So it may not work for new applications which embed addresses in payloads.

This mechanism supports unicast communications only. In order to support multicast functions, some other additional functionalities must be considered in the function mapper module.

Since the IPv6 API has new advanced features, it is difficult to translate such kinds of IPv6 APIs into IPv4 APIs. Thus, IPv6 inbound communication with advanced features may be discarded.

7. Security Considerations

The security consideration of BIA mostly relies on that of [NAT-PT]. The differences are due to the address translation occurring at the API and not in the network layer. That is, since the mechanism uses the API translator at the socket API level, hosts can utilize the security of the network layer (e.g., IPsec) when they communicate with IPv6 hosts using IPv4 applications via the mechanism. As well, there isn't a DNS ALG as in NAT-PT, so there is no interference with DNSSEC.

The use of address pooling may open a denial of service attack vulnerability. So BIA should employ the same sort of protection techniques as [NAT-PT] does.

8. Acknowledgments

We would like to acknowledge the implementation contributions by Wanjik Lee (wjlee@arang.miryang.ac.kr) and i2soft Corporation (www.i2soft.net).

9. References

[TRANS-MECH] Gilligan, R. and E. Nordmark, "Transition Mechanisms for IPv6 Hosts and Routers", RFC 2893, August 2000.

[SIIT] Nordmark, E., "Stateless IP/ICMP Translator (SIIT)", RFC 2765, February 2000.

[FTP] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, October 1985.

[NAT] Srisuresh, P. and K. Egevang, "Traditional IP Network Address Translator (Traditional NAT)", RFC 3022, January 2001.

[IPV4] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.

[IPV6] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[NAT-PT] Tsirtsis, G. and P. Srisuresh, "Network Address Translation - Protocol Translation (NAT-PT)", RFC 2766, February 2000.

[BIS] Tsuchiya, K., Higuchi, H. and Y. Atarashi, "Dual Stack Hosts using the "Bump-In-the-Stack" Technique (BIS)", RFC 2767, February 2000.

[SOCK-EXT] Gilligan, R., Thomson, S., Bound, J. and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999.

[RFC 2119] Bradner S., "Key words for use in RFCs to indicate Requirement Levels", RFC 2119, March 1997.

Appendix A : API list intercepted by BIA

The following functions are the API list which SHOULD be intercepted by BIA module.

The functions that the application uses to pass addresses into the system are:

```
bind()
connect()
sendmsg()
sendto()
```

The functions that return an address from the system to an application are:

```
accept()
recvfrom()
recvmsg()
getpeername()
getsockname()
```

The functions that are related to socket options are:

```
getsocketopt()
setsocketopt()
```

The functions that are used for conversion of IP addresses embedded in application layer protocol (e.g., FTP, DNS, etc.) are:

```
recv()
send()
read()
write()
```

As well, raw sockets for IPv4 and IPv6 MAY be intercepted.

Most of the socket functions require a pointer to the socket address structure as an argument. Each IPv4 argument is mapped into corresponding an IPv6 argument, and vice versa.

According to [SOCK-EXT], the following new IPv6 basic APIs and structures are required.

IPv4	new IPv6
AF_INET	AF_INET6
sockaddr_in	sockaddr_in6
gethostbyname()	getaddrinfo()
gethostbyaddr()	getnameinfo()
inet_ntoa()/inet_addr()	inet_pton()/inet_ntop()
INADDR_ANY	in6addr_any

BIA MAY intercept inet_ntoa() and inet_addr() and use the address mapper for those. Doing that enables BIA to support literal IP addresses.

The gethostbyname() call return a list of addresses. When the name resolver function invokes getaddrinfo() and getaddrinfo() returns multiple IP addresses, whether IPv4 or IPv6, they SHOULD all be represented in the addresses returned by gethostbyname(). Thus if getaddrinfo() returns multiple IPv6 addresses, this implies that multiple address mappings will be created; one for each IPv6 address.

Authors' Addresses

Seungyun Lee
ETRI PEC
161 Kajong-Dong, Yusong-Gu, Taejon 305-350, Korea
Tel: +82 42 860 5508
Fax: +82 42 861 5404
EMail: syl@pec.etri.re.kr

Myung-Ki Shin
ETRI PEC
161 Kajong-Dong, Yusong-Gu, Taejon 305-350, Korea
Tel: +82 42 860 4847
Fax: +82 42 861 5404
EMail: mkshin@pec.etri.re.kr

Yong-Jin Kim
ETRI
161 Kajong-Dong, Yusong-Gu, Taejon 305-350, Korea
Tel: +82 42 860 6564
Fax: +82 42 861 1033
EMail: yjkim@pec.etri.re.kr

Alain Durand
Sun Microsystems, inc.
25 Network circle
Menlo Park, CA 94025, USA
Fax: +1 650 786 5896
EMail: Alain.Durand@sun.com

Erik Nordmark
Sun Microsystems Laboratories
180, avenue de l'Europe
38334 SAINT ISMIER Cedex, France
Tel: +33 (0)4 76 18 88 03
Fax: +33 (0)4 76 18 88 88
EMail: erik.nordmark@sun.com

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.