Network Working Group
Request for Comments: 4038
Category: Informational

M-K. Shin, Ed.
ETRI/NIST
Y-G. Hong
ETRI
J. Hagino
IIJ
P. Savola
CSC/FUNET
E. M. Castro
GSYC/URJC

March 2005

Application Aspects of IPv6 Transition

Status of This Memo

Copyright Notice

Abstract

As IPv6 networks are deployed and the network transition is discussed, one should also consider how to enable IPv6 support in applications running on IPv6 hosts, and the best strategy to develop IP protocol support in applications. This document specifies scenarios and aspects of application transition. It also proposes guidelines on how to develop IP version-independent applications during the transition period.

Table of Contents

1. Introduction

As IPv6 is introduced in the IPv4-based Internet, several general issues will arise, such as routing, addressing, DNS, and scenarios.

An important key to a successful IPv6 transition is compatibility with the large installed base of IPv4 hosts and routers. This issue has already been extensively studied, and work is still in progress. [2893BIS] describes the basic transition mechanisms: dual-stack deployment and tunneling. Various other kinds of mechanisms have been developed for the transition to an IPv6 network. However, these transition mechanisms take no stance on whether applications support IPv6.

This document specifies application aspects of IPv6 transition. Two inter-related topics are covered:

1. How different network transition techniques affect applications, and strategies for applications to support IPv6 and IPv4.

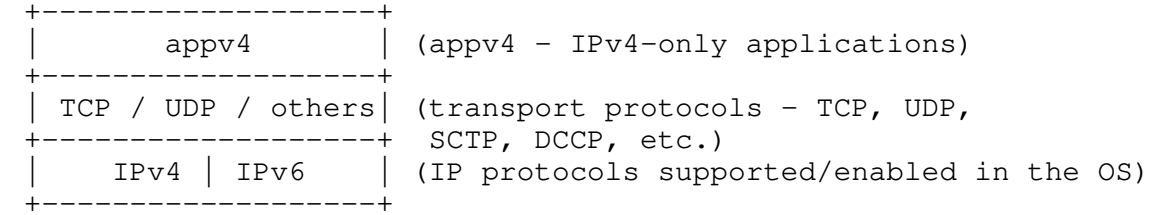
2. How to develop IPv6-capable or protocol-independent applications ("application porting guidelines") using standard APIs [RFC3493][RFC3542].

In the context of this document, the term "application" covers all kinds of applications, but the focus is on those network applications which have been developed using relatively low-level APIs (such as the "C" language, using standard libraries). Many such applications could be command-line driven, but that is not a requirement.

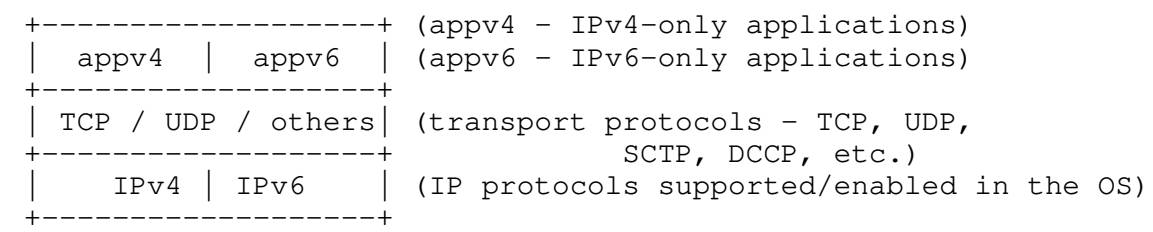
Applications will have to be modified to support IPv6 (and IPv4) by using one of a number of techniques described in sections 2 - 4. Guidelines for developing such applications are presented in sections 5 and 6.

2. Overview of IPv6 Application Transition

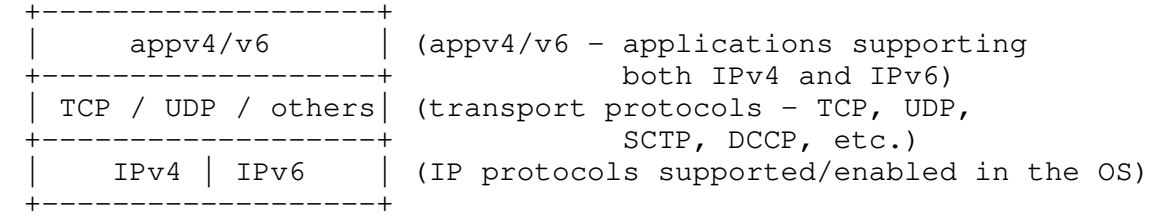
The transition of an application can be classified by using four different cases (excluding the first case when there is no IPv6 support in either the application or the operating system):



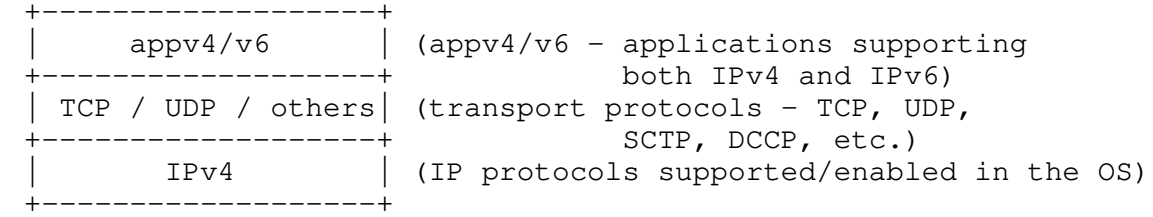
Case 1. IPv4 applications in a dual-stack node.



Case 2. IPv4-only applications and IPv6-only applications
in a dual-stack node.



Case 3. Applications supporting both IPv4 and IPv6
in a dual-stack node.



Case 4. Applications supporting both IPv4 and IPv6
in an IPv4-only node.

Figure 1. Overview of Application Transition

Figure 1 shows the cases of application transition.

Case 1: IPv4-only applications in a dual-stack node.
IPv6 protocol is introduced in a node, but applications are not yet ported to support IPv6.

Case 2: IPv4-only applications and IPv6-only applications in a dual-stack node.
Applications are ported for IPv6-only. Therefore there are two similar applications, one for each protocol version (e.g., ping and ping6).

Case 3: Applications supporting both IPv4 and IPv6 in a dual stack node.
Applications are ported for both IPv4 and IPv6 support. Therefore, the existing IPv4 applications can be removed.

Case 4: Applications supporting both IPv4 and IPv6 in an IPv4-only node.
Applications are ported for both IPv4 and IPv6 support, but the same applications may also have to work when IPv6 is not being used (e.g., disabled from the OS).

The first two cases are not interesting in the longer term; only few applications are inherently IPv4- or IPv6-specific, and should work with both protocols without having to care about which one is being used.

3. Problems with IPv6 Application Transition

There are several reasons why the transition period between IPv4 and IPv6 applications may not be straightforward. These issues are described in this section.

3.1. IPv6 Support in the OS and Applications Are Unrelated

Considering the cases described in the previous section, IPv4 and IPv6 protocol stacks are likely to co-exist in a node for a long time.

Similarly, most applications are expected to be able to handle both IPv4 and IPv6 during another long period. A dual-stack operating system is not intended to have both IPv4 and IPv6 applications. Therefore, IPv6-capable application transition may be independent of protocol stacks in a node.

Applications capable of both IPv4 and IPv6 will probably have to work properly in IPv4-only nodes (whether the IPv6 protocol is completely disabled or there is no IPv6 connectivity at all).

3.2. DNS Does Not Indicate Which IP Version Will Be Used

In a node, the DNS name resolver gathers the list of destination addresses. DNS queries and responses are sent by using either IPv4 or IPv6 to carry the queries, regardless of the protocol version of the data records [DNSTRANS].

The DNS name resolution issue related to application transition is that by only doing a DNS name lookup a client application can not be certain of the version of the peer application. For example, if a server application does not support IPv6 yet but runs on a dual-stack machine for other IPv6 services, and this host is listed with an AAAA record in the DNS, the client application will fail to connect to the server application. This is caused by a mismatch between the DNS query result (i.e., IPv6 addresses) and a server application version (i.e., IPv4).

Using SRV records would avoid these problems. Unfortunately, they are not used widely enough to be applicable in most cases. Hence an operational solution is to use "service names" in the DNS. If a node offers multiple services, but only some of them over IPv6, a DNS name may be added for each of these services or group of services (with the associated A/AAAA records), not just a single name for the physical machine, also including the AAAA records. However, the applications cannot depend on this operational practice.

The application should request all IP addresses without address family constraints and try all the records returned from the DNS, in some order, until a working address is found. In particular, the application has to be able to handle all IP versions returned from the DNS. This issue is discussed in more detail in [DNSOPV6].

3.3. Supporting Many Versions of an Application is Difficult

During the application transition period, system administrators may have various versions of the same application (an IPv4-only application, an IPv6-only application, or an application supporting both IPv4 and IPv6).

Typically one cannot know which IP versions must be supported prior to doing a DNS lookup *and* trying (see section 3.2) the addresses returned. Therefore if multiple versions of the same application are available, the local users have difficulty selecting the right version supporting the exact IP version required.

To avoid problems with one application not supporting the specified protocol version, it is desirable to have hybrid applications supporting both.

An alternative approach for local client applications could be to have a "wrapper application" that performs certain tasks (such as figuring out which protocol version will be used) and calls the IPv4/IPv6-only applications as necessary. This application would perform connection establishment (or similar tasks) and pass the opened socket to another application. However, as applications such as this would have to do more than just perform a DNS lookup or determine the literal IP address given, they will become complex -- likely much more so than a hybrid application. Furthermore, writing "wrapping" applications that perform complex operations with IP addresses (such as FTP clients) might be even more challenging or even impossible. In short, wrapper applications do not look like a robust approach for application transition.

4. Description of Transition Scenarios and Guidelines

Once the IPv6 network is deployed, applications supporting IPv6 can use IPv6 network services to establish IPv6 connections. However, upgrading every node to IPv6 at the same time is not feasible, and transition from IPv4 to IPv6 will be a gradual process.

Dual-stack nodes provide one solution to maintaining IPv4 compatibility in unicast communications. In this section we will analyze different application transition scenarios (as introduced in section 2) and guidelines for maintaining interoperability between applications running in different types of nodes.

Note that the first two cases, IPv4-only and IPv6-only applications, are not interesting in the longer term; only few applications are inherently IPv4- or IPv6-specific, and should work with both protocols without having to care about which one is being used.

4.1. IPv4 Applications in a Dual-Stack Node

In this scenario, the IPv6 protocol is added in a node, but IPv6-capable applications aren't yet available or installed. Although the node implements the dual stack, IPv4 applications can only manage IPv4 communications and accept/establish connections from/to nodes that implement an IPv4 stack.

To allow an application to communicate with other nodes using IPv6, the first priority is to port applications to IPv6.

In some cases (e.g., when no source code is available), existing IPv4 applications can work if the Bump-in-the-Stack [BIS] or Bump-in-the-API [BIA] mechanism is installed in the node. We strongly recommend that application developers not use these mechanisms when application source code is available. Also, they should not be used as an excuse not to port software or to delay porting.

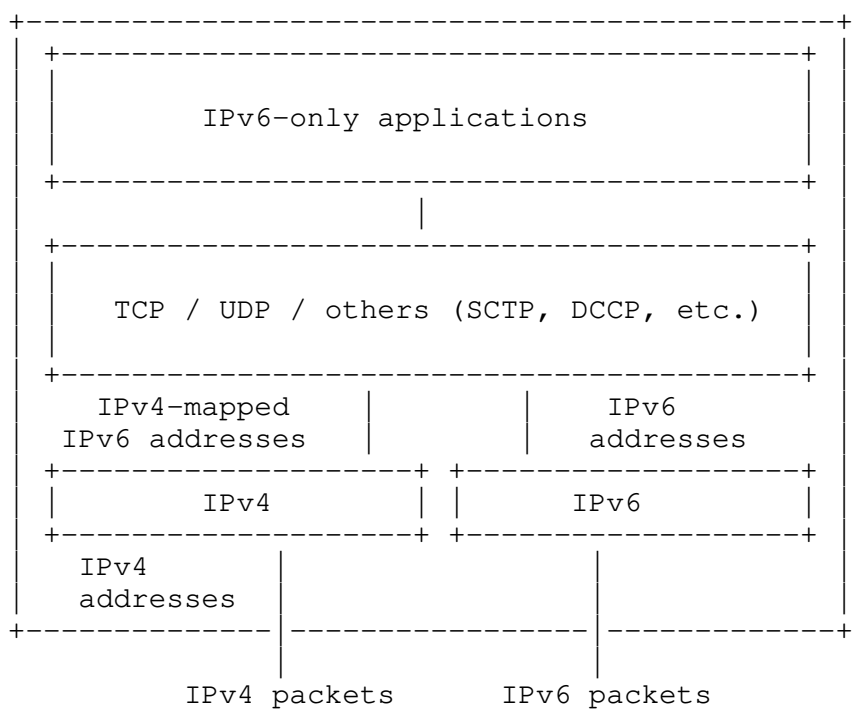
When [BIA] or [BIS] is used, the problem described in section 3.2 arises - (the IPv4 client in a [BIS]/[BIA] node tries to connect to an IPv4 server in a dual stack system). However, one can rely on the [BIA]/[BIS] mechanism, which should cycle through all the addresses instead of applications.

[BIS] and [BIA] do not work with all kinds of applications - in particular, with applications that exchange IP addresses as application data (e.g., FTP). These mechanisms provide IPv4 temporary addresses to the applications and locally make a translation between IPv4 and IPv6 communication. Therefore, these IPv4 temporary addresses are only valid in the node scope.

4.2. IPv6 Applications in a Dual-Stack Node

As we have seen in the previous section, applications should be ported to IPv6. The easiest way to port an IPv4 application is to substitute the old IPv4 API references with the new IPv6 APIs with one-to-one mapping. This way the application will be IPv6-only. This IPv6-only source code cannot work in IPv4-only nodes, so the old IPv4 application should be maintained in these nodes. This necessitates having two similar applications working with different protocol versions, depending on the node they are running (e.g., telnet and telnet6). This case is undesirable, as maintaining two versions of the same source code per application could be difficult. This approach would also cause problems for users having to select which version of the application to use, as described in section 3.3.

Most implementations of dual stack allow IPv6-only applications to interoperate with both IPv4 and IPv6 nodes. IPv4 packets going to IPv6 applications on a dual-stack node reach their destination because their addresses are mapped by using IPv4-mapped IPv6 addresses: the IPv6 address ::FFFF:x.y.z.w represents the IPv4 address x.y.z.w.



We will analyze the behaviour of IPv6-applications that exchange IPv4 packets with IPv4 applications by using the client/server model. We consider the default case to be when the IPV6_V6ONLY socket option has not been set. In these dual-stack nodes, this default behavior allows a limited amount of IPv4 communication using the IPv4-mapped IPv6 addresses.

IPv6-only server:
: When an IPv4 client application sends data to an IPv6-only server application running on a dual-stack node by using the wildcard address, the IPv4 client address is interpreted as the IPv4-mapped IPv6 address in the dual-stack node. This allows the IPv6 application to manage the communication. The IPv6 server will use this mapped address as if it were a regular IPv6 address, and a usual IPv6 connection. However, IPv4 packets will be exchanged between the nodes. Kernels with dual stack properly interpret IPv4-mapped IPv6 addresses as IPv4 ones, and vice versa.

IPv6-only client:
: IPv6-only client applications in a dual-stack node will not receive IPv4-mapped addresses from the hostname resolution API functions unless a special hint, AI_V4MAPPED, is given. If it

is, the IPv6 client will use the returned mapped address as if it were a regular IPv6 address, and a usual IPv6 connection. However, IPv4 packets will be exchanged between applications.

Respectively, with IPV6_V6ONLY set, an IPv6-only server application will only communicate with IPv6 nodes, and an IPv6-only client only with IPv6 servers, as the mapped addresses have been disabled. This option could be useful if applications use new IPv6 features such as Flow Label. If communication with IPv4 is needed, either IPV6_V6ONLY must not be used, or dual-stack applications must be used, as described in section 4.3.

Some implementations of dual-stack do not allow IPv4-mapped IPv6 addresses to be used for interoperability between IPv4 and IPv6 applications. In these cases, there are two ways to handle the problem:

1. Deploy two different versions of the application (possibly attached with '6' in the name).

2. Deploy just one application supporting both protocol versions as described in the next section.

The first method is not recommended because of a significant number of problems associated with selecting the right applications. These problems are described in sections 3.2 and 3.3.

Therefore, there are two distinct cases to consider when writing one application to support both protocols:

1. Whether the application can (or should) support both IPv4 and IPv6 through IPv4-mapped IPv6 addresses or the applications should support both explicitly (see section 4.3), and

2. Whether the systems in which the applications are used support IPv6 (see section 4.4).

Note that some systems will disable (by default) support for internal IPv4-mapped IPv6 addresses. The security concerns regarding these are legitimate, but disabling them internally breaks one transition mechanism for server applications originally written to bind() and listen() to a single socket by using a wildcard address. This forces the software developer to rewrite the daemon to create two separate sockets, one for IPv4 only and the other for IPv6 only, and then to use select(). However, mapping-enabling of IPv4 addresses on any particular system is controlled by the OS owner and not necessarily

by a developer. This complicates developers' work, as they now have to rewrite the daemon network code to handle both environments, even for the same OS.

4.3. IPv4/IPv6 Applications in a Dual-Stack Node

Applications should be ported to support both IPv4 and IPv6. Over time, the existing IPv4-only applications could be removed. As we have only one version of each application, the source code will typically be easy to maintain and to modify, and there are no problems managing which application to select for which communication.

This transition case is the most advisable. During the IPv6 transition period, applications supporting both IPv4 and IPv6 should be able to communicate with other applications, irrespective of the version of the protocol stack or the application in the node. Dual applications allow more interoperability between heterogeneous applications and nodes.

If the source code is written in a protocol-independent way, without dependencies on either IPv4 or IPv6, applications will be able to communicate with any combination of applications and types of nodes.

Implementations typically prefer IPv6 by default if the remote node and application support it. However, if IPv6 connections fail, version-independent applications will automatically try IPv4 ones. The resolver returns a list of valid addresses for the remote node, and applications can iterate through all of them until connection succeeds.

Application writers should be aware of this protocol ordering, which is typically the default, but the applications themselves usually need not be [RFC3484].

If the source code is written in a protocol-dependent way, the application will support IPv4 and IPv6 explicitly by using two separate sockets. Note that there are some differences in bind() implementation - that is, in whether one can first bind to IPv6 wildcard addresses, and then to those for IPv4. Writing applications that cope with this can be a pain. Implementing IPV6_V6ONLY simplifies this. The IPv4 wildcard bind fails on some systems because the IPv4 address space is embedded into IPv6 address space when IPv4-mapped IPv6 addresses are used.

A more detailed porting guideline is described in section 6.

4.4. IPv4/IPv6 Applications in an IPv4-Only Node

As the transition is likely to take place over a longer time frame, applications already ported to support both IPv4 and IPv6 may be run on IPv4-only nodes. This would typically be done to avoid supporting two application versions for older and newer operating systems, or to support a case in which the user wants to disable IPv6 for some reason.

The most important case is the application support on systems where IPv6 support can be dynamically enabled or disabled by the users. Applications on such a system should be able to handle a situation IPv6 would not be enabled. Another scenario is when an application is deployed on older systems that do not support IPv6 at all (even the basic APIs such as getaddrinfo). In this case, the application designer has to make a case-by-case judgment call as to whether it makes sense to have compile-time toggle between an older and a newer API (having to support both in the code), or whether to provide getaddrinfo etc. function support on older platforms as part of the application libraries.

Depending on application/operating system support, some may want to ignore this case, but usually no assumptions can be made, and applications should also work in this scenario.

An example is an application that issues a socket() command, first trying AF_INET6 and then AF_INET. However, if the kernel does not have IPv6 support, the call will result in an EPROTONOSUPPORT or EAFNOSUPPORT error. Typically, errors like these lead to exiting the socket loop, and AF_INET will not even be tried. The application will need to handle this case or build the loop so that errors are ignored until the last address family.

This case is just an extension of the IPv4/IPv6 support in the previous case, covering one relatively common but often-ignored case.

5. Application Porting Considerations

The minimum changes for IPv4 applications to work with IPv6 are based on the different size and format of IPv4 and IPv6 addresses.

Applications have been developed with IPv4 network protocol in mind. This assumption has resulted in many IP dependencies through source code.

The following list summarizes the more common IP version dependencies in applications:

a) Presentation format for an IP address: An ASCII string that represents the IP address, a dotted-decimal string for IPv4, and a hexadecimal string for IPv6.

b) Transport layer API: Functions to establish communications and to exchange information.

c) Name and address resolution: Conversion functions between hostnames and IP addresses.

d) Specific IP dependencies: More specific IP version dependencies, such as IP address selection, application framing, and storage of IP addresses.

e) Multicast applications: One must find the IPv6 equivalents to the IPv4 multicast addresses and use the right socket configuration options.

The following subsections describe the problems with the aforementioned IP version dependencies. Although application source code can be ported to IPv6 with minimum changes related to IP addresses, some recommendations are given to modify the source code in a protocol-independent way, which will allow applications to work with both IPv4 and IPv6.

5.1. Presentation Format for an IP Address

Many applications use IP addresses to identify network nodes and to establish connections to destination addresses. For instance, using the client/server model, clients usually need an IP address as an application parameter to connect to a server. This IP address is usually provided in the presentation format, as a string. There are two problems when porting the presentation format for an IP address: the allocated memory and the management of the presentation format.

Usually, the memory allocated to contain an IPv4 address representation as a string is unable to contain an IPv6 address. Applications should be modified to prevent buffer overflows made possible by the larger IPv6 address.

IPv4 and IPv6 do not use the same presentation format. IPv4 uses a dot (.) to separate the four octets written in decimal notation, and IPv6 uses a colon (:) to separate each pair of octets written in hexadecimal notation [RFC3513]. In cases where one must be able to specify, for example, port numbers with the address (see below), it may be desirable to require placing the address inside the square brackets [TextRep].

A particular problem with IP address parsers comes when the input is actually a combination of IP address and port number. With IPv4 these are often coupled with a colon; for example, "192.0.2.1:80". However, this approach would be ambiguous with IPv6, as colons are already used to structure the address.

Therefore, the IP address parsers that take the port number separated with a colon should distinguish IPv6 addresses somehow. One way is to enclose the address in brackets, as is done with Uniform Resource Locators (URLs) [RFC2732]; for example, http://[2001:db8::1]:80.

Some applications also need to specify IPv6 prefixes and lengths: The prefix length should be inserted outside of the square brackets, if used; for example, [2001:db8::]/64 or 2001:db8::/64 and not [2001:db8::/64]. Note that prefix/length notation is syntactically indistinguishable from a legal URI; therefore, the prefix/length notation must not be used when it isn't clear from the context that it's used to specify the prefix and length and not, for example, a URI.

In some specific cases, it may be necessary to give a zone identifier as part of the address; for example, fe80::1%eth0. In general, applications should not need to parse these identifiers.

The IP address parsers should support enclosing the IPv6 address in brackets, even when the address is not used in conjunction with a port number. Requiring that the user always give a literal IP address enclosed in brackets is not recommended.

Note that some applications may also represent IPv6 address literals differently; for example, SMTP [RFC2821] uses [IPv6:2001:db8::1].

Note that the use of address literals is strongly discouraged for general-purpose direct input to the applications. Host names and DNS should be used instead.

5.2. Transport Layer API

Communication applications often include a transport module that establishes communications. Usually this module manages everything related to communications and uses a transport-layer API, typically as a network library. When an application is ported to IPv6, most changes should be made in this application transport module in order to be adapted to the new IPv6 API.

In the general case, porting an existing application to IPv6 requires an examination of the following issues related to the API:

- Network Information Storage: IP address Data Structures
  The new structures must contain 128-bit IP addresses. The use of generic address structures, which can store any address family, is recommended.

  Sometimes special addresses are hard-coded in the application source code. Developers should pay attention to these in order to use the new address format. Some of these special IP addresses are wildcard local, loopback, and broadcast. IPv6 does not have the broadcast addresses, so applications can use multicast instead.

- Address Conversion Functions
  The address conversion functions convert the binary address representation to the presentation format and vice versa. The new conversion functions are specified to the IPv6 address format.

- Communication API Functions
  These functions manage communications. Their signatures are defined based on a generic socket address structure. The same functions are valid for IPv6; however, the IP address data structures used when calling these functions require the updates.

- Network Configuration Options
  These are used when different communication models are configured for Input/Output (I/O) operations (blocking/nonblocking, I/O multiplexing, etc.) and should be translated for IPv6.

5.3. Name and Address Resolution

From the application point of view, the name and address resolution is a system-independent process. An application calls functions in a system library, the resolver, which is linked into the application when it is built. However, these functions use IP address structures, that are protocol dependent and must be reviewed to support the new IPv6 resolution calls.

With IPv6, there are two new basic resolution functions, getaddrinfo() and getnameinfo(). The first returns a list of all configured IP addresses for a hostname. These queries can be constrained to one protocol family; for instance, only IPv4 or only

IPv6 addresses. However, it is recommended that all configured IP addresses be obtained to allow applications to work with every kind of node. The second function returns the hostname associated to an IP address.

5.4. Specific IP Dependencies

5.4.1. IP Address Selection

Unlike the IPv4 model, IPv6 promotes the configuration of multiple IP addresses per node, however, applications only use a destination/source pair for a communication. Choosing the right IP source and destination addresses is a key factor that may determine the route of IP datagrams.

Typically, nodes, not applications, automatically solve the source address selection. A node will choose the source address for a communication following some rules of best choice, per [RFC3484], but will also allow applications to make changes in the ordering rules.

When selecting the destination address, applications usually ask a resolver for the destination IP address. The resolver returns a set of valid IP addresses from a hostname. Unless applications have a specific reason to select any particular destination address, they should try each element in the list until the communication succeeds.

In some cases, the application may need to specify its source address. The destination address selection process picks the best destination for the source address (instead of picking the best source address for the chosen destination address). Note that if it is not yet known which protocol will be used for communication there may be an increase in complexity for IP version - independent applications that have to specify the source address (especially for client applications. Fortunately, specifying the source address is not typically required).

5.4.2. Application Framing

The Application Level Framing (ALF) architecture controls mechanisms that traditionally fall within the transport layer. Applications implementing ALF are often responsible for packetizing data into Application Data Units (ADUs). The application problem with ALF arrives from the ADU size selection to obtain better performance.

Applications using connectionless protocols (such as UDP) typically need application framing. These applications have three choices: (1) to use packet sizes no larger than the IPv6 minimum Maximum Transmission Unit (MTU) of 1280 bytes [RFC2460], (2) to use any

packet sizes, but to force IPv6 fragmentation/reassembly when necessary, or (3) to optimize the packet size and avoid unnecessary fragmentation/reassembly, and to guess or find out the optimal packet sizes that can be sent and received, end-to-end, on the network. This memo takes no stance on that approach is best.

Note that the most optimal ALF depends on dynamic factors such as Path MTU or whether IPv4 or IPv6 is being used (due to different header sizes, possible IPv6-in-IPv4 tunneling overhead, etc.). These factors have to be taken into consideration when application framing is implemented.

5.4.3. Storage of IP Addresses

Some applications store IP addresses as remote peer information. For instance, one of the most popular ways to register remote nodes in collaborative applications uses IP addresses as registry keys.

Although the source code that stores IP addresses can be modified to IPv6 by following the previous basic porting recommendations, applications should not store IP addresses for the following reasons:

- IP addresses can change throughout time; for instance, after a renumbering process.

- The same node can reach a destination host using different IP addresses, possibly with a different protocol version.

When possible, applications should store names such as FQDNs or other protocol-independent identities instead of addresses. In this case applications are only bound to specific addresses at run time, or for the duration of a cache lifetime. Other types of applications, such as massive peer-to-peer systems with their own rendezvous and discovery mechanisms, may need to cache addresses for performance reasons, but cached addresses should not be treated as permanent, reliable information. In highly dynamic networks, any form of name resolution may be impossible, and here again addresses must be cached.

5.5. Multicast Applications

There is an additional problem in porting multicast applications. When multicast facilities are used some changes must be carried out to support IPv6. First, applications must change the IPv4 multicast addresses to IPv6 ones, and second, the socket configuration options must be changed.

All IPv6 multicast addresses encode scope; the scope was only implicit in IPv4 (with multicast groups in 239/8). Also, although a large number of application-specific multicast addresses have been assigned with IPv4, this has been (luckily enough) avoided with IPv6. So there are no direct equivalents for all the multicast addresses. For link-local multicast, it's possible to pick almost anything within the link-local scope. The global groups could use unicast prefix - based addresses [RFC3306]. All in all, this may force the application developers to write more protocol-dependent code.

Another problem is that IPv6 multicast does not yet have a standardized mechanism for traditional Any Source Multicast for Interdomain multicast. The models for Any Source Multicast (ASM) or Source-Specific Multicast (SSM) are generally similar between IPv4 and IPv6, but it is possible that PIM-SSM will become more widely deployed in IPv6 due to its simpler architecture.

It might be beneficial to port the applications to use SSM semantics, requiring off-band source discovery mechanisms and a different API [RFC3678]. Inter-domain ASM service is available only through a method embedding the Rendezvous Point address in the multicast address [Embed-RP].

Another generic problem with multiparty conferencing applications, similar to the issues with peer-to-peer applications, is that all users of the session must use the same protocol version (IPv4 or IPv6), or some form of proxy or translator (e.g., [MUL-GW]).

6. Developing IP Version - Independent Applications

As stated, dual applications working with both IPv4 and IPv6 are recommended. These applications should avoid IP dependencies in the source code. However, if IP dependencies are required, one of the better solutions would be to build a communication library that provides an IP version - independent API to applications and that hides all dependencies.

To develop IP version - independent applications, the following guidelines should be considered.

6.1. IP Version - Independent Structures

All memory structures and APIs should be IP version-independent. One should avoid structs in_addr, in6_addr, sockaddr_in, and sockaddr_in6.

Suppose a network address is passed to some function, foo(). If one uses struct in_addr or struct in6_addr, results an extra parameter to indicate address family, as below:

```
struct in_addr in4addr;
struct in6_addr in6addr;
 /* IPv4 case */
foo(&in4addr, AF_INET);
 /* IPv6 case */
foo(&in6addr, AF_INET6);
```

This leads to duplicated code and having to consider each scenario from both perspectives independently, which is difficult to maintain. So we should use struct sockaddr_storage, as below:

```
struct sockaddr_storage ss;
int sslen;
/* AF independent! - use sockaddr when passing a pointer */
/* note: it's typically necessary to also pass the length
   explicitly */
foo((struct sockaddr *)&ss, sslen);
```

6.2. IP Version - Independent APIs

The new address independent variants getaddrinfo() and getnameinfo() hide the gory details of name-to-address and address-to-name translations. They implement functionalities of the following functions:

```
gethostbyname()
gethostbyaddr()
getservbyname()
getservbyport()
```

They also obsolete the functionality of gethostbyname2(), defined in [RFC2133].

The new variants can perform hostname/address and service name/port lookups, though the features can be turned off, if desired. Getaddrinfo() can return multiple addresses, as below:

```
localhost.      IN A    127.0.0.1
                IN A    127.0.0.2
                IN AAAA ::1
```

In this example, if IPv6 is preferred, getaddrinfo first returns ::1; then both 127.0.0.1 and 127.0.0.2 are in a random order.

Getaddrinfo() and getnameinfo() can query hostname and service name/port at once.

Hardcoding AF-dependent knowledge is not preferred in the program. Constructs such as that below should be avoided:

```
    /* BAD EXAMPLE */
    switch (sa->sa_family) {
    case AF_INET:
            salen = sizeof(struct sockaddr_in);
            break;
   }
```

Instead, we should use the ai_addrlen member of the addrinfo structure, as returned by getaddrinfo().

The gethostbyname(), gethostbyaddr(), getservbyname(), and getservbyport() are mainly used to get server and client sockets. In the following sections, we will see simple examples creating these sockets by using the new IPv6 resolution functions.

6.2.1. Example of Overly Simplistic TCP Server Application

A simple TCP server socket at service name (or port number string) SERVICE:

```
   /*
    * BAD EXAMPLE: does not implement the getaddrinfo loop as
    * specified in 6.3.  This may result in one of the following:
    *  - an IPv6 server, listening at the wildcard address,
    *    allowing IPv4 addresses through IPv4-mapped IPv6 addresses.
    *  - an IPv4 server, if IPv6 is not enabled,
    *  - an IPv6-only server, if IPv6 is enabled but IPv4-mapped IPv6
    *    addresses are not used by default, or
    *  - no server at all, if getaddrinfo supports IPv6, but the
    *    system doesn't, and socket(AF_INET6, ...) exits with an
    *    error.
    */
   struct addrinfo hints, *res;
   int error, sockfd;

   memset(&hints, 0, sizeof(hints));
   hints.ai_flags = AI_PASSIVE;
   hints.ai_family = AF_UNSPEC;
   hints.ai_socktype = SOCK_STREAM;
   error = getaddrinfo(NULL, SERVICE, &hints, &res);
   if (error != 0) {
      /* handle getaddrinfo error */
```

```
}

sockfd = socket(res->family, res->ai_socktype, res->ai_protocol);
if (sockfd < 0) {
   /* handle socket error */
}

if (bind(sockfd, res->ai_addr, res->ai_addrlen) < 0) {
   /* handle bind error */
}

/* ... */

freeaddrinfo(res);
```

6.2.2. Example of Overly Simplistic TCP Client Application

A simple TCP client socket connecting to a server running at node name (or IP address presentation format) SERVER_NODE and service name (or port number string) SERVICE follows:

```
/*
 * BAD EXAMPLE: does not implement the getaddrinfo loop as
 * specified in 6.3.  This may result in one of the following:
 *  - an IPv4 connection to an IPv4 destination,
 *  - an IPv6 connection to an IPv6 destination,
 *  - an attempt to try to reach an IPv6 destination (if AAAA
 *    record found), but failing -- without fallbacks -- because:
 *     o getaddrinfo supports IPv6 but the system does not
 *     o IPv6 routing doesn't exist, so falling back to e.g., TCP
 *       timeouts
 *     o IPv6 server reached, but service not IPv6-enabled or
 *       firewalled away
 *  - if the first destination is not reached, there is no
 *    fallback to the next records
 */
struct addrinfo hints, *res;
int error, sockfd;

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(SERVER_NODE, SERVICE, &hints, &res);
if (error != 0) {
     /* handle getaddrinfo error */
}
```

```
sockfd = socket(res->family, res->ai_socktype, res->ai_protocol);
if (sockfd < 0) {
     /* handle socket error */
}

if (connect(sockfd, res->ai_addr, res->ai_addrlen) < 0 ) {
     /* handle connect error */
}

/* ... */

freeaddrinfo(res);
```

6.2.3. Binary/Presentation Format Conversion

We should consider the binary and presentation address format conversion APIs. The following functions convert network address structure in its presentation address format and vice versa:

```
inet_ntop()
inet_pton()
```

Both are from the basic socket extensions for IPv6. However, these conversion functions are protocol-dependent. It is better to use getnameinfo()/getaddrinfo() (inet_pton and inet_ntop equivalents are described in Appendix A).

Conversion from network address structure to presentation format can be written as follows:

```
struct sockaddr_storage ss;
char addrStr[INET6_ADDRSTRLEN];
char servStr[NI_MAXSERV];
int error;

/* fill ss structure */

error = getnameinfo((struct sockaddr *)&ss, sizeof(ss),
                    addrStr, sizeof(addrStr),
                    servStr, sizeof(servStr),
                    NI_NUMERICHOST);
```

Conversions from presentation format to network address structure can be written as follows:

```
struct addrinfo hints, *res;
char addrStr[INET6_ADDRSTRLEN];
int error;

/* fill addrStr buffer */

memset(&hints, 0, sizeof(hints));
hints.ai_family = AF_UNSPEC;

error = getaddrinfo(addrStr, NULL, &hints, &res);
if (error != 0) {
    /* handle getaddrinfo error */
}

/* res->ai_addr contains the network address structure */
/* ... */
freeaddrinfo(res);
```

6.3. Iterated Jobs for Finding the Working Address

In a client code, when multiple addresses are returned from getaddrinfo(), we should try all of them until connection succeeds. When a failure occurs with socket(), connect(), bind(), or some other function, the code should go on to try the next address.

In addition, if something is wrong with the socket call because the address family is not supported (i.e., in case of section 4.4), applications should try the next address structure.

Note: In the following examples, the socket() return value error handling could be simplified by always continuing on with the socket loop instead of performing special checking of specific error numbers.

6.3.1. Example of TCP Server Application

The previous TCP server example should be written as follows:

```
#define MAXSOCK 2
struct addrinfo hints, *res;
int error, sockfd[MAXSOCK], nsock=0;

memset(&hints, 0, sizeof(hints));
hints.ai_flags = AI_PASSIVE;
hints.ai_family = AF_UNSPEC;
```

```
hints.ai_socktype = SOCK_STREAM;

error = getaddrinfo(NULL, SERVICE, &hints, &res);
if (error != 0) {
    /* handle getaddrinfo error */
}

for (aip=res; aip && nsock < MAXSOCK; aip=aip->ai_next) {
    sockfd[nsock] = socket(aip->ai_family,
                           aip->ai_socktype,
                           aip->ai_protocol);

    if (sockfd[nsock] < 0) {
        switch errno {
             case EAFNOSUPPORT:
             case EPROTONOSUPPORT:
                 /*
                  *  e.g., skip the errors until
                  *  the last address family,
                  *  see section 4.4.
                  */
                  if (aip->ai_next)
                          continue;

                  else {
                         /* handle unknown protocol errors */
                          break;
                  }
             default:
                  /* handle other socket errors */
                  ;
         }

    } else {
        int on = 1;
        /* optional: works better if dual-binding to wildcard
           address */
        if (aip->ai_family == AF_INET6) {
            setsockopt(sockfd[nsock], IPPROTO_IPV6, IPV6_V6ONLY,
                       (char *)&on, sizeof(on));
            /* errors are ignored */
        }
        if (bind(sockfd[nsock], aip->ai_addr,
                                aip->ai_addrlen) < 0 ) {
            /* handle bind error */
            close(sockfd[nsock]);
            continue;
        }
```

```
            if (listen(sockfd[nsock], SOMAXCONN) < 0) {
                /* handle listen errors */
                close(sockfd[nsock]);
                continue;
            }
        }
        nsock++;
    }
    freeaddrinfo(res);

    /* check that we were able to obtain the sockets */
```

6.3.2. Example of TCP Client Application

The previous TCP client example should be written as follows:

```
    struct addrinfo hints, *res, *aip;
    int sockfd, error;

    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;

    error = getaddrinfo(SERVER_NODE, SERVICE, &hints, &res);
    if (error != 0) {
        /* handle getaddrinfo error */
    }

    for (aip=res; aip; aip=aip->ai_next) {

        sockfd = socket(aip->ai_family,
                        aip->ai_socktype,
                        aip->ai_protocol);

        if (sockfd < 0) {
            switch errno {
                 case EAFNOSUPPORT:
                 case EPROTONOSUPPORT:
                     /*
                      *  e.g., skip the errors until
                      *  the last address family,
                      *  see section 4.4.
                      */
                      if (aip->ai_next)
                              continue;
                      else {
                             /* handle unknown protocol errors */
                              break;
```

```
                        }

                    default:
                        /* handle other socket errors */
                        ;
                }

            } else {
                if (connect(sockfd, aip->ai_addr, aip->ai_addrlen) == 0)
                    break;

                /* handle connect errors */
                close(sockfd);
                sockfd=-1;
            }
        }

        if (sockfd > 0) {
            /* socket connected to server address */

            /* ... */
        }

        freeaddrinfo(res);
```

7. Transition Mechanism Considerations

The mechanism [NAT-PT] introduces a special set of addresses, formed of an NAT-PT prefix and an IPv4 address these refer to IPv4 addresses translated by NAT-PT DNS-ALG. In some cases, one might be tempted to handle these differently.

However, IPv6 applications must not be required to distinguish "normal" and "NAT-PT translated" addresses (or any other kind of special addresses, including the IPv4-mapped IPv6 addresses): This would be completely impractical, and if the distinction must be made, it must be done elsewhere (e.g., kernel, system libraries).

8. Security Considerations

There are a number of security considerations for IPv6 transition, but those are outside the scope of this memo.

To ensure the availability and robustness of the service even when transitioning to IPv6, this memo describes a number of ways to make applications more resistant to failures by cycling through addresses until a working one is found. Doing this properly is critical to maintain availability and to avoid loss of service.

A special consideration about application transition is how IPv4-mapped IPv6 addresses are handled. The use in the API can be seen both as a merit (easier application transition) and as a burden (difficulty in ensuring whether the use was legitimate). Note that some systems will disable (by default) support for internal IPv4-mapped IPv6 addresses. The security concerns regarding these on the wire are legitimate, but disabling it internally breaks one transition mechanism for server applications originally written to bind() and listen() to a single socket by using a wildcard address [V6MAPPED]. This should be considered in more detail when applications are designed.

9. Acknowledgments

Some of guidelines for development of IP version-independent applications (section 6) were first brought up by [AF-APP]. Other work to document application porting guidelines has also been in progress; for example, [IP-GGF] and [PRT]. We would like to thank the members of the v6ops working group and the application area for helpful comments. Special thanks are due to Brian E. Carpenter, Antonio Querubin, Stig Venaas, Chirayu Patel, Jordi Palet, and Jason Lin for extensive review of this document. We acknowledge Ron Pike for proofreading the document.

10. References

10.1. Normative References

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, May 2003.

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

[BIA] Lee, S., Shin, M-K., Kim, Y-J., Nordmark, E., and A. Durand, "Dual Stack Hosts Using "Bump-in-the-API" (BIA)", RFC 3338, October 2002.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

[RFC3484] Draves, R., "Default Address Selection for Internet Protocol version 6 (IPv6)", RFC 3484, February 2003.

[RFC3513] Hinden, R. and S. Deering, "Internet Protocol Version 6 (IPv6) Addressing Architecture", RFC 3513, April 2003.

10.2. Informative References

[2893BIS] Nordmark, E. and R. E. Gilligan, "Basic Transition Mechanisms for IPv6 Hosts and Routers", Work in Progress, June 2004.

[RFC2133] Gilligan, R., Thomson, S., Bound, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2133, April 1997.

[RFC2732] Hinden, R., Carpenter, B., and L. Masinter, "Format for Literal IPv6 Addresses in URL's", RFC 2732, December 1999.

[RFC2821] Klensin, J., "Simple Mail Transfer Protocol", RFC 2821, April 2001.

[TextRep] Main, A., "Textual Representation of IPv4 and IPv6 Addresses", Work in Progress, October 2003.

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

[DNSTRANS] Durand, A. and J. Ihren, "DNS IPv6 Transport Operational Guidelines", BCP 91, RFC 3901, September 2004.

[DNSOPV6] Durand, A., Ihren, J. and P. Savola, "Operational Considerations and Issues with IPv6 DNS", Work in Progress, May 2004.

[AF-APP] Hagino, J., "Implementing AF-independent application", http://www.kame.net/newsletter/19980604/, 2001.

[V6MAPPED] Hagino, J., "IPv4 mapped address considered harmful", Work in Progress, April 2002.

[IP-GGF] Chown, T., Bound, J., Jiang, S. and P. O'Hanlon, "Guidelines for IP version independence in GGF specifications", Global Grid Forum(GGF) Documentation, work in Progress, September 2003.

[Embed-RP] Savola, P. and B. Haberman, "Embedding the Rendezvous Point (RP) Address in an IPv6 Multicast Address", RFC 3956, November 2004.

[RFC3306] Haberman, B. and D. Thaler, "Unicast-Prefix-based IPv6 Multicast Addresses", RFC 3306, August 2002.

[RFC3678] Thaler, D., Fenner, B., and B. Quinn, "Socket Interface Extensions for Multicast Source Filters, RFC 3678, January 2004.

[MUL-GW] Venaas, S., "An IPv4 - IPv6 multicast gateway", Work in Progress, February 2003.

[PRT] Castro, E. M., "Programming guidelines on transition to IPv6 LONG project", Work in Progress, January 2003.

Appendix A. Other Binary/Presentation Format Conversions

Section 6.2.3 describes the preferred way to perform binary/presentation format conversions; these can also be done by using inet_pton() and inet_ntop() and by writing protocol-dependent code. This approach is not recommended, but it is provided here for reference and comparison.

Note that inet_ntop()/inet_pton() lose the scope identifier (if used, e.g., with link-local addresses) in the conversions, contrary to the getaddrinfo()/getnameinfo() functions.

A.1. Binary to Presentation Using inet_ntop()

Conversions from network address structure to presentation format can be written as follows:

```
struct sockaddr_storage ss;
char addrStr[INET6_ADDRSTRLEN];

/* fill ss structure */

switch (ss.ss_family) {

     case AF_INET:
          inet_ntop(ss.ss_family,
                    &((struct sockaddr_in *)&ss)->sin_addr,
                    addrStr,
                    sizeof(addrStr));
          break;

     case AF_INET6:
          inet_ntop(ss.ss_family,
                     &((struct sockaddr_in6 *)&ss)->sin6_addr,
                     addrStr,
                     sizeof(addrStr));

          break;

     default:
          /* handle unknown family */
}
```

Note that, the destination buffer addrStr should be long enough to contain the presentation address format: INET_ADDRSTRLEN for IPv4 and INET6_ADDRSTRLEN for IPv6. As INET6_ADDRSTRLEN is longer than INET_ADDRSTRLEN, the first one is used as the destination buffer length.

A.2. Presentation to Binary Using inet_pton()

Conversions from presentation format to network address structure can be written as follows:

```
struct sockaddr_storage ss;
struct sockaddr_in *sin;
struct sockaddr_in6 *sin6;
char addrStr[INET6_ADDRSTRLEN];

/* fill addrStr buffer and ss.ss_family */

switch (ss.ss_family) {
      case AF_INET:
            sin = (struct sockaddr_in *)&ss;
            inet_pton(ss.ss_family,
                      addrStr,
                      (sockaddr *)&sin->sin_addr));
            break;

      case AF_INET6:
            sin6 = (struct sockaddr_in6 *)&ss;
            inet_pton(ss.ss_family,
                      addrStr,
                      (sockaddr *)&sin6->sin6_addr);
            break;

      default:
          /* handle unknown family */
}
```

Note that, the address family of the presentation format must be known.

Authors' Addresses

Myung-Ki Shin
ETRI/NIST
820 West Diamond Avenue
Gaithersburg, MD 20899, USA

Phone: +1 301 975-3613
Fax: +1 301 590-0932
EMail: mshin@nist.gov

Yong-Guen Hong
ETRI PEC
161 Gajeong-Dong, Yuseong-Gu, Daejeon 305-350, Korea

Phone: +82 42 860 6447
Fax: +82 42 861 5404
EMail: yghong@pec.etri.re.kr

Jun-ichiro itojun HAGINO
Research Laboratory, Internet Initiative Japan Inc.
Takebashi Yasuda Bldg.,
3-13 Kanda Nishiki-cho,
Chiyoda-ku,Tokyo 101-0054, JAPAN

Phone: +81-3-5259-6350
Fax: +81-3-5259-6351
EMail: itojun@iijlab.net

Pekka Savola
CSC/FUNET
Espoo, Finland

EMail: psavola@funet.fi

Eva M. Castro
Rey Juan Carlos University (URJC)
Departamento de Informatica, Estadistica y Telematica
C/Tulipan s/n
28933 Madrid - SPAIN

EMail: eva@gsyc.escet.urjc.es

Full Copyright Statement

Intellectual Property

Acknowledgement

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