v6ops Working Group
INTERNET DRAFT
Expires: December 2004

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

June 2004

Application Aspects of IPv6 Transition
<draft-ietf-v6ops-application-transition-03.txt>

Status of this Memo

Abstract

As IPv6 networks are deployed and the network transition 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 arise such as routing, addressing, DNS, scenarios, etc.

One important key to a successful IPv6 transition is the compatibility with the large installed base of IPv4 hosts and routers. This issue had been already been extensively studied, and the work is still in progress. In particular, [2893BIS] describes the basic transition mechanisms, dual-stack deployment and tunneling. In addition, various kinds of transition mechanisms have been developed for the transition to an IPv6 network. However, these transition mechanisms take no stance on whether applications support IPv6 or not.

This document specifies application aspects of IPv6 transition. That is, two inter-related topics are covered:

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

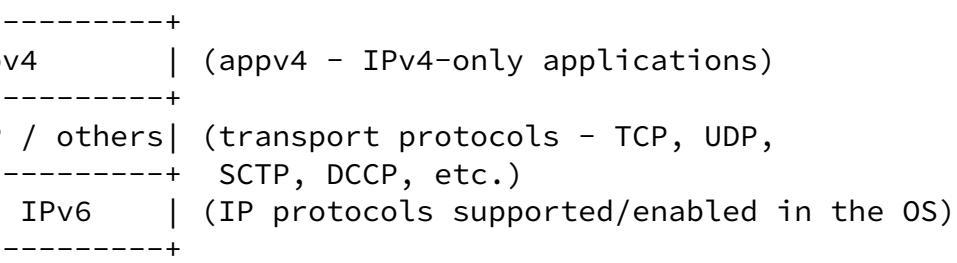
2. How to develop IPv6-capable or protocol-independent applications ("application porting guidelines").

Applications will need to be modified to support IPv6 (and IPv4),

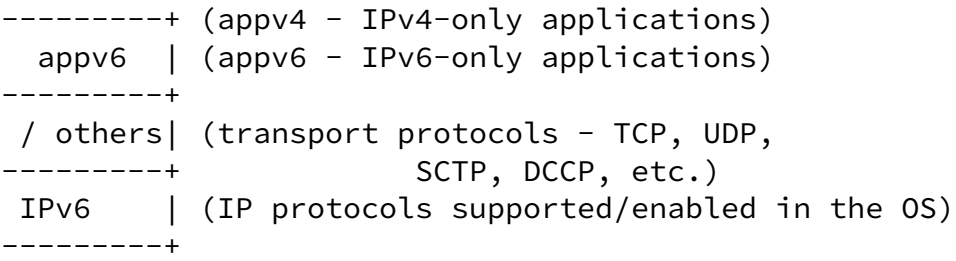
using one of a number of techniques described in sections 2-4. Some guidelines to develop such application are then presented in sections 5 and 6.

2. Overview of IPv6 Application Transition

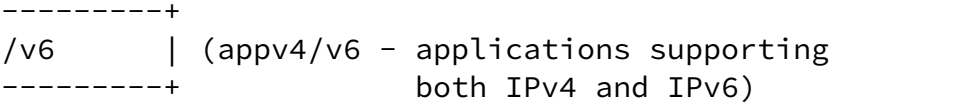
The transition of an application can be classifed using four different cases (excluding the first case when there is no IPv6 support either in the application or the operating system), as follows:



Case 1. IPv4 applications in a dual-stack node



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



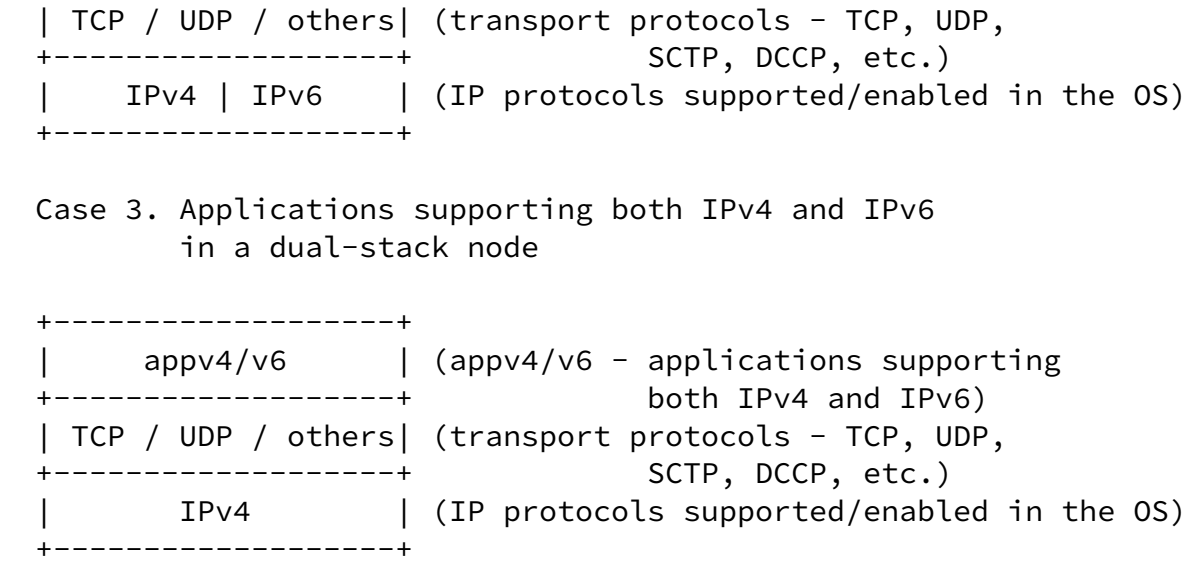


Figure 1. Overview of Application Transition

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).

Note that this draft does not address DCCP and SCTP considerations at this phase.

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 in a node is likely to co-exist for a long time.

Similarly, most applications are expected to be able to handle both IPv4 and IPv6 during another, unrelated long time period. That is, the operating system being dual stack does not mean having both IPv4 and IPv6 applications. Therefore, IPv6-capable application transition may be independent of protocol stacks in a node.

It is even probable that applications capable of both IPv4 and IPv6 will 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

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

The issue of DNS name resolution related to application transition, is that a client application can not be certain of the version of the peer application by only doing a DNS name lookup. 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 a AAAA record in the DNS, the client application will fail to connect to the server application. This is caused by a mis-match 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 sufficiently widely used to be applicable in most cases. Hence an operational technique is to use "service names" in the DNS, that is, if a node is offering multiple services, but only some of them over IPv6, add a DNS name for each of these services (with the associated A/AAAA records), not just a single name for the whole node, also including the AAAA records. However, the applications cannot depend on such operational practices.

In consequence, 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, the local users have a difficulty selecting the right application version supporting the exact IP version required if multiple versions of the same application are available.

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

One could argue that an alternative approach for local client applications could be to have a "wrapper application" which performs certain tasks (like figures out which protocol version will be used) and calls the IPv4/IPv6-only applications as necessary. In other words, such applications would perform connection establishment (or similar), and pass the opened socket

to the other application. However, as these applications would have to do more than just perform a DNS lookup or figure out the literal IP address given, they will get complex -- likely much more complex than writing a hybrid application. Furthermore, "wrapping" applications which perform complex operations with IP addresses (like FTP clients) might be even more challenging or even impossible. In summary, wrapper applications does 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 and 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 are one of the ways to maintain IPv4 compatibility in unicast communications. In this section we will analyze different application transition scenarios (as introduced in section 2) and guidelines to maintain interoperability between applications running in different types of nodes.

4.1 IPv4 Applications in a Dual-stack Node

This scenario happens if 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. Then, IPv4 applications can only accept/establish connections from/to nodes which implement an IPv4 stack.

In order 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. 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 sould not use these mechanisms when application source code is available. Also, it should not be used as an excuse not to port software or delay porting.

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

[BIS] or [BIA] does not work with all kinds of applications. In particular, the applications which 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. Hence, these IPv4 temporary addresses are only valid in the node scope."

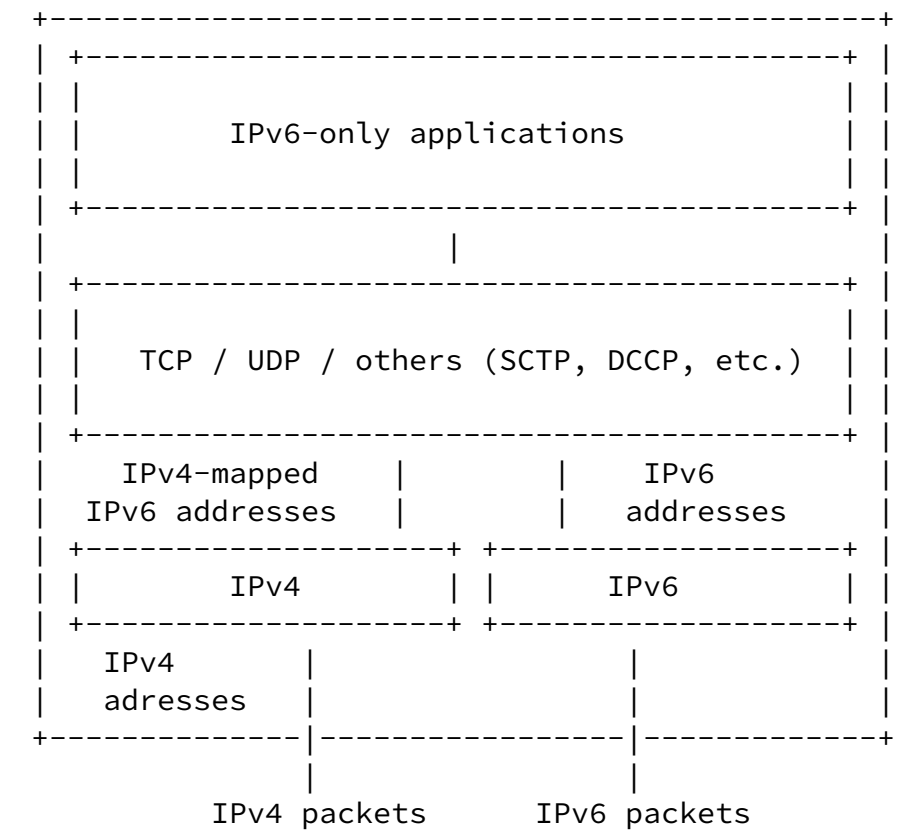
4.2 IPv6 Applications in a Dual-stack Node

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 can not work in IPv4-only nodes, so the old IPv4 application should be maintained in these nodes. Then, we will get two similar applications working with different protocol versions, depending on the node they are running (e.g., telnet and telnet6). This case is undesirable since maintaining two versions of the same source code per application could be a difficult task. In addition, this approach would cause problems for the users when 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 to IPv6 ones 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 which exchange IPv4 packets with IPv4 applications using the client/server model. We consider the default case when the IPV6_V6ONLY socket option has not been set. This default behavior of IPv6 applications in these dual-stack nodes 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 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 get IPv4-mapped addresses from the hostname resolution API functions unless a special hint, AI_V4MAPPED, is given. If given, the IPv6 client will use the returned mapped address as if it were a regular IPv6 address, and a usual IPv6 connection. However, again 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 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 be used, as described in section 4.3.

There are some implementations of dual-stack which do not allow IPv4-mapped IPv6 addresses to be used for interoperability between IPv4 and IPv6 applications. In that case, there are two ways to handle the problem:

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

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 amount of problems associated with selecting the right applications. This problems are described in sections 3.2 and 3.3.

Therefore, there are actually 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 should the applications support both explicitly (see section 4.3), and

2. whether the systems where the applications are used support IPv6 at all or not (see section 4.4).

Note that some systems will disable (by default) support for internal IPv4-mapped IPv6 addresses. The security concerns regarding IPv4-mapped IPv6 addresses on the wire are legitimate but disabling it internally breaks one transition mechanism for server applications which were originally written to bind() and listen() to a single socket using a wildcard address. This forces the software developer to rewrite the daemon to create 2 separate sockets, one for IPv4 only and the other for IPv6 only, and then use select(). However, enabling mapping of IPv4 addresses on any particular system is controlled by the OS owner and not necessarilly by a developer. This complicates the developer's work as he now has 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; such applications are sometimes called IP version-independent applications. After that, the existing IPv4-only applications could be removed. Since we have only one version of each application, the source code will be typically 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 versions 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 by-default prefer IPv6 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.

Applications writers should be aware of this typical by-default ordering, but the applications themselves typically need not be aware of the the local protocol ordering [RFC 3484].

If the source code is written in a protocol-dependent way, the application will support IPv4 and IPv6 explicitly using 2 separate sockets. Note that there are some differences in bind() implementation, whether you can first bind to the IPv6, and then

IPv4, wildcard addresses. It can be a pain to write applications that cope with this. If IPV6_V6ONLY is implemented, this becomes simpler. The reason the IPv4 wildcard bind fails on some systems is that the IPv4 address space is embedded into IPv6 address space when using IPv4-mapped IPv6 addresses.

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 happen over a longer timeframe, applications that have already been ported to support both IPv4 and IPv6 may be run on IPv4-only nodes. This would typically be done to avoid having to support two application versions for older and newer operating systems, or to support the case that 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 the situation where IPv6 would not be enabled. The secondary scenario is when an application could be deployed on older systems which do not support IPv6 at all (even the basic getaddrinfo etc. APIs). In that case the application designer has to make a case-by-case judgement call whether it makes sense to have compile-time toggle between an older and 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 how application/operating system support is done, 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, encountering errors like these leads 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 in such a way that errors are ignored until the last address family.

So, 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 to IPv4 applications to work with IPv6 are based on the different size and format of IPv4 and IPv6 addresses.

Applications have been developed with the assumption they would use IPv4 as their network protocol. This assumption results 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: it is an ASCII string which represents the IP address, dotted-decimal string for IPv4 and 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, and vice versa.

d) Specific IP dependencies: more specific IP version dependencies, such as: IP address selection,

application framing, 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.

In the following subsections, the problems with the aforementioned IP version dependencies are analyzed. 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 using 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 allocated memory 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 [RFC 3513]. In cases where it one must be able to specify e.g., 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 such as "192.0.2.1:80". However, such an approach would be ambiguous with IPv6 as colons are already used to structure the address.

Therefore, the IP address parsers which take the port number separated with a colon should represent IPv6 addresses somehow. One way is to enclose the address in brackets, as is done with Uniform Resource Locators (URLs) [RFC 2732], like 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, like [2001:db8::]/64 or 2001:db8::/64 -- not for example [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 e.g., a URI.

In some specific cases, it may be necessary to give a zone identifier as part of the address, like 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 it's not used in conjunction with a port number, but requiring that the user always gives a literal IP address enclosed in brackets is not recommended.

One should note that some applications may also represent IPv6 address literals differently; for example, SMTP [RFC 2821] 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 porting an application 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 them 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.
  They are used when configuring different communication models for Input/Output (I/O) operations (blocking/nonblocking, I/O multiplexing, etc.) and should be translated to the IPv6 ones.

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 this is built. However, these functions use IP address structures, which are protocol dependent, and must be reviewed to support the new IPv6 resolution calls.

There are two basic resolution functions. The first function 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, the recommendation is

that all configured IP addresses should be obtained to allow applications to work with every kind of node. And the second function returns the hostname associated to an IP address.

5.4 Specific IP Dependencies

5.4.1 IP Address Selection

IPv6 promotes the configuration of multiple IP addresses per node, which is a difference when compared with the IPv4 model; 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, [RFC 3484], but also allowing 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 just try each element in the list until the communication succeeds.

In some cases, the application may need to specify its source address. Then 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 there may be an increase in complexity for IP-version independent applications which have to specify the source address (especially for client applications; fortunately, specifying the source address is not typically required), if it is not yet known which protocol will be used for communication.

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 when using ALF is the ADU size selection to obtain better performance.

Application framing is typically needed by applications using connectionless protocols (such as UDP). Such applications have three choices: (1) to use packet sizes no larger than IPv6 IPv6 minimum Maximum Transmission Unit (MTU) of 1280 bytes [RFC2460], (2) to use whatever packet sizes but force IPv6 fragmentation/reassembly when necessary, or (3) to optimize the packet size and avoid unnecessary fragmentation/reassembly, guess or find out the optimal packet sizes which can be sent and

received, end-to-end, on the network. This memo takes no stance on which approach to adopt.

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 have to be taken into consideration when implementing application framing.

5.4.3 Storage of IP Addresses

Some applications store IP addresses as information of remote peers. For instance, one of the most popular ways to register remote nodes in collaborative applications is based on using IP addresses as registry keys.

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

- 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 storing 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 using multicast facilities 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 the IPv6 multicast addresses encode scope; the scope was only implicit in IPv4 (with multicast groups in 239/8). Also, while a large number of application-specific multicast addresses have been assigned with IPv4, this has been (luckily enough) avoided in 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 [RFC 3306]. All in all, this may force the application developers to write more protocol dependent code.

Another problem is/has been 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.

So, it might be beneficial to port the applications to use SSM semantics, requiring off-band source discovery mechanisms and the use of a different API [RFC 3678]. Inter-domain ASM service is available only through a method embedding the Rendezvous Point address in the multicast address [Embed-RP].

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

6. Developing IP version-independent Applications

As we have seen before, 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 best solutions is based on building a communication library which provides an IP version independent API to applications and hides all dependencies.

In order to develop IP version independent applications, the following guidelines should be considered.

6.1 IP version-independent Structures

All of the memory structures and APIs should be IP version-independent. In that sense, one should avoid structs in_addr, in6_addr, sockaddr_in and sockaddr_in6.

Suppose you pass a network address to some function, foo(). If you use struct in_addr or struct in6_addr, you will end up with 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);
```

However, this leads to duplicated code and having to consider each scenario from both perspectives independently; this is difficult to maintain. So, we should use struct sockaddr_storage like 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

getaddrinfo() and getnameinfo() are new address independent variants that 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].

These can perform hostname/address and service name/port lookups, though the features can be turned off if desirable. 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 returns first ::1, and then both 127.0.0.1 and 127.0.0.2 is in a random order.

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

It is not preferred to hardcode AF-dependent knowledge into the program. The construct like 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. Following, we will see simple examples to create these sockets 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 which is running at node name (or IP address presentation format) SERVER_NODE and 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 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

In addition, 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; instead it is better to use getnameinfo()/getaddrinfo() as follows (inet_pton and inet_ntop equivalents are described in Appendix A).

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

```
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 simplied by substituting special checking of specific error numbers by always continuing on with the socket loop.

6.3.1 Example of TCP Server Application

The previous example TCP server example should be written:

```
#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:

```
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

A mechanism, [NAT-PT], introduces a special set of addresses, formed of NAT-PT prefix and an IPv4 address; this refers 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): that would be completely impractical, and if such distinction must be made, it must be done elsewhere (e.g. kernel, system libraries).

8. Security Considerations

There are a number of security considerations with 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 described 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 avoid unavailability and loss of service.

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

9. Acknowledgements

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 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

Normative References

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

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

[BIS] K. Tsuchiya, H. Higuchi, Y. Atarashi, "Dual Stack Hosts

using the "Bump-In-the-Stack" Technique (BIS)," RFC 2767, February 2000.

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

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

[RFC 3484] R. Draves, "Default Address Selection for IPv6," RFC 3484, February 2003.

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

Informative References

[2893BIS] E. Nordmark, R. E. Gilligan, "Basic Transition Mechanisms for IPv6 Hosts and Routers," <draft-ietf-v6ops-mech-v2-03.txt>, June 2004, Work-in-progress.

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

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

[TextRep] A. Main, "Textual Representation of IPv4 and IPv6 Addresses," <draft-main-ipaddr-text-rep-01.txt>, Oct 2003, Work in Progress.

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

[DNSTRANS] A. Durand, J. Ihren, "DNS IPv6 transport operational guidelines," <draft-ietf-dnsop-ipv6-transport-guidelines-02.txt>, March 2004, Work in Progress.

[DNSOPV6] A. Durand, J. Ihren, P. Savola, "Operational Considerations and Issues with IPv6 DNS," <draft-ietf-dnsop-ipv6-dns-issues-07.txt>, May 2004, Work in Progress.

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

[V6MAPPED] J. Hagino, "IPv4 mapped address considered harmful", <draft-itojun-v6ops-v4mapped-harmful-02.txt>, Apr 2002, Work in Progress.

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

[Embed-RP] P. Savola, B. Haberman, "Embedding the Address of RP in IPv6 Multicast Address," <draft-ietf-mboned-embeddedrp-00.txt>, October 2003, Work in Progress.

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

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

[MUL-GW] S. Venaas, "An IPv4 - IPv6 multicast gateway," <draft-venaas-mboned-v4v6mcastgw-00.txt>, February 2003, Work in Progress.

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

Authors' Addresses

Myung-Ki Shin
ETRI/NIST
820 West Diamond Avenue
Gaithersburg, MD 20899, USA
Tel : +1 301 975-3613
Fax : +1 301 590-0932
E-mail : mshin@nist.gov

Yong-Guen Hong
ETRI PEC
161 Gajeong-Dong, Yuseong-Gu, Daejeon 305-350, Korea
Tel : +82 42 860 6447

Fax : +82 42 861 5404
E-mail : 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
Tel: +81-3-5259-6350

Fax: +81-3-5259-6351
E-mail: itojun@iijlab.net

Pekka Savola
CSC/FUNET
Espoo, Finland
E-mail: psavola@funet.fi

Eva M. Castro
Rey Juan Carlos University (URJC)
Departamento de Informatica, Estadistica y Telematica
C/Tulipan s/n
28933 Madrid - SPAIN
E-mail: eva@gsyc.escet.urjc.es

Appendix A. Other binary/Presentation Format Conversions

Section 6.2.3 described the preferred way of performing binary/presentation format conversions; these can also be done using inet_pton() and inet_ntop() by writing protocol-dependent code. This is not recommended, but 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:

```
 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, the destination buffer addrStr should be long enough to contain the presentation address format: INET_ADDRSTRLEN for IPv4 and INET6_ADDRSTRLEN for IPv6. Since 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, the address family of the presentation format must be known.

Intellectual Property Statement

it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in BCP 78 and BCP 79.

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at http://www.ietf.org/ipr.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

Copyright Statement

Acknowledgment

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