Migrating to IPv6: The Problem of the Applications

Fulvio Risso (fulvio.risso@polito.it),

Dipartimento di Automatica e Informatica, Politecnico di Torino
Corso Duca degli Abruzzi, 24 – 10129 Torino – ITALY
Phone: +39-011-564.7008
Fax: +39-011-564.7099

***Abstract*—After several years of studies and experiments, the days of the new IP, identified as version 6, are coming; the growing pressure is mostly due to address shortage (especially in Far East countries), the imminent introduction of new 3G mobile devices, and routing scalability problems. While most of the modern operating systems and network devices (routers) are already IPv6-capable, one of the biggest problems is related to the huge amount of work involving application migration. This paper studies some issues related to the migration of applications to IPv6: (*i*) how difficult the porting is and (*ii*) how to make an application IPv6-compatible when its source code is not available. This paper presents solutions to the latter problem that "retrofit" off-the-shelf server applications. Such solutions can also be deployed in response to the pressing demand to promptly make native IPv6 service available to IPv6-enabled clients.**

Keywords: *IPv6 applications, IPv6 socket interface, RFC 3493.*

I. Introduction

IPv6 is the new network-layer protocol poised to replace the old IPv4, defined about 30 years ago. IPv6 was born in the mid-90s to cope with address space shortage. Massive deployment of private addressing and address translation and proxy mechanisms introduced while waiting for IPv6 specification and implementations to be available, has decreased the urgency to introduce the new protocol. However, in the last few years the pressure to adopt it has increased, mostly due to address shortage (especially in Far East countries), the future introduction of new 3G mobile devices, and routing scalability problems. Furthermore, the U.S. Department of Defense has recently decided that all its new hardware and software must support IPv6, which is a big boost for the adoption of this protocol.

In recent years we have seen huge efforts from operating systems and network devices companies to introduce IPv6 support into their products. Although some issues remain to be solved, we can consider IPv6 support in operating systems as an almost completed task. Modern operating systems (BSD, Linux, Solaris, Windows XP) have excellent IPv6 stacks, although some of the applications provided with them are still IPv4-only. From the network device manufacturers (mostly router vendors) point of view, most of the devices can be equipped with an IPv6 stack. Most vendors (notably Cisco, Juniper, Extreme Networks) offer acceptable support for IPv6, although some of the devices are still implementing IPv6 forwarding fully in software, while IPv4 packets are handled by specialized hardware. Moreover, some helper protocols are still missing.

Having achieved a reasonable support in operating systems and network infrastructure, the next step is the migration of applications. This is the most difficult task because of the huge amount of network-based applications, which means tons of code that has to be changed — hence a large number of programmers involved. Such effort is comparable to the one related to the Y2k (Year 2000) problem only a few years ago, when most applications had to be verified.

This paper explores the problem of modifying applications to make them compatible with IPv6. Section II describes and provides a quantitative evaluation of the efforts required to insert IPv6 support into an application, provided that the source code is available. A few special cases are presented in Section III, specifically applications based on some external framework (e.g. Java and .NET) and applications that must compile on platforms with or without IPv6 support. Section IV focuses on applications on which source code is not available, particularly servers. For instance, a typical corporate information system includes both applications custom developed in-house and off-the-shelf software, and the process of making them IPv6 compatible is different in these two cases. Section V presents the case study referred to a University information system whose most important services have been modified in order to accept native IPv6 connections. Finally, Section VI summarizes the work and gives some conclusive remarks.

II. Migrating Applications Through Source Code Modifications

Although this seems to be the easiest case, the modification of the application source code requires a lot of work in order to support both IPv4 and IPv6 (i.e. dual-stack applications). For instance, RFC 3493 (Basic Socket Interface Extensions for IPv6) [1] proposes some limited changes to the BSD socket API, but the devil lays into the details. For instance, most of the "standard" system calls (e.g. `socket()`, `connect()`, `bind()`, `accept()`, `send()`, `recv()`, and more) remain unchanged. However, often the programmer has to retrieve the required parameters in a different way; therefore the code is to be modified.

The comparison of the code in Figure 1 (written in `C` language according to the traditional API syntax) and Figure 2 (written according to the new syntax defined in RFC 3493) may be an excellent example. Both code fragments refer to a server socket during its opening phase and the binding of all local addresses. As shown in Figure 1, the programmer specifies the address family (`AF_INET`, i.e. IPv4) directly into the `socket()` call. This procedure changes in Figure 2 because the programmer cannot know if the server has an IPv4 stack, an IPv6 one, or both. To avoid this problem, the programmer uses the `getaddrinfo()` function that returns a linked list of `addrinfo` structures, which hold the list of addresses we can bind to. Then, the programmer can open a socket waiting on one of the addresses (either IPv4 or IPv6) available on the machine. Although the prototype of the `socket()` function does not change, the code around it looks pretty different. This means that the source code must be visually inspected and adapted where needed.

From the programmer's point of view, details matter. For instance, even the code related to the `bind()` function must be changed because the old-programming style uses a variable of type `sockaddr_in` as a parameter. Vice versa, the new programming style uses some members of the `addrinfo` structure, returned by the `getaddrinfo()`. Furthermore, the old programming style defines the port number as a number, while the new one defines the same value as a string. These are only a few examples of the differences between the old BSD API and the new one defined in RFC 3493.

The function `getaddrinfo()` has been introduced in the new socket API and plays a key role in providing protocol independence. This function is, perhaps, the most important change in the socket API.

```
#define PORT 2000

void server ()
{
int SockDescr;                      // Descriptor for the network socket
struct sockaddr_in SockAddr;        // Address of the server socket descr.

   if ( (SockDescr= socket(AF_INET, SOCK_STREAM, 0)) < 0 )
   {
      error("Server: cannot open socket.");
      return;
   }

   memset(& SockAddr, 0, sizeof(SockAddr));
   SockAddr.sin_family = AF_INET;
   SockAddr.sin_addr.s_addr= htonl(INADDR_ANY); /* all local addresses */
   SockAddr.sin_port = htons(PORT);    /* Convert to network byte order */

   if (bind(SockDescr, (struct sockaddr *) &SockAddr, sizeof(SockAddr)) < 0)
   {
      error("Server: bind failure");
      return;
   }

   /* Other code follows */
```

Figure 1. Opening a server socket with the traditional socket API.

```
#define PORT "2000"

void server ()
{
int SockDescr;                      // Descriptor for the network socket
struct addrinfo Hints, *AddrInfo;   // Helper structures

   memset(&Hints, 0, sizeof(Hints));
   Hints.ai_family = AF_UNSPEC; // or AF_INET / AF_INET6
   Hints.ai_socktype = SOCK_STREAM;
   Hints.ai_flags = AI_PASSIVE;      /* ready to a bind() socket */

   if (getaddrinfo(NULL /* all local addr */, PORT, Hints, AddrInfo) != 0)
   {
      error("Server: cannot resolve Address / Port ");
      return;
   }

   // Open a socket with the correct address family for this address.
   if ( (SockDescr= socket(AddrInfo->ai_family,
                    AddrInfo->ai_socktype, AddrInfo->ai_protocol)) < 0 )
   {
      error("Server: cannot open socket.");
      return;
   }

   if (bind(SockDescr, AddrInfo->ai_addr, AddrInfo->ai_addrlen) < 0)
   {
      error("Server: bind failure");
      return;
   }

   /* Other code follows */
```

Figure 2. Opening a server socket with the new RFC 3493 API.

A. Identifying the changes

We realized the porting of two open-source applications – Freeamp and Gnucleus – in order to understand and quantify the efforts required to transform an IPv4-based application into a dual stack one. FreeAmp is a free MP3 player, available for Windows and most of the Unix platforms, which plays MP3 files received across the network from a stream server. Gnucleus is a peer-to-peer file sharing application based on the Gnutella protocol and is available for Windows only.

These applications use the socket interface basically for three different tasks: address resolution, session establishment and data reception. The code related to the first point has been modified by making use of the `getaddrinfo()` function,

Function	IPv4	IPv6	Changes needed	Lines of code factor	FreeAmp			Gnucleus		
					# of calls	Total lines of code	Changed lines of code	# of calls	Total lines of code	Changed lines of code
socket()	Y	Y	Y	1	10	10	10	1	1	1
connect()	Y	Y	Y	2	8	16	16	1	2	2
bind()	Y	Y	Y	2	4	8	8	1	2	2
listen()	Y	Y	N	1	-	-	-	1	1	-
accept()	Y	Y	Y	1	-	-	-	1	1	1
close()	Y	Y	N	1	20	20	-	1	1	-
gethostbyname()	Y	Y	Y	4	8	32	32	2	8	8
getpeername()	Y	Y	Y	4	-	-	-	1	4	4
getsockname()	Y	Y	Y	4	-	-	-	1	4	4
inet_ntoa()	Y	N	Y	1	-	-	-	3	3	3
inet_addr()	Y	N	Y	1	12	12	12	3	3	3
read()	Y	Y	N	1	2	2	-	-	-	-
write()	Y	Y	N	1	2	2	-	-	-	-
send()	Y	Y	N	1	6	6	-	1	1	-
sendto()	Y	Y	Y	1	-	-	-	1	1	1
recv()	Y	Y	N	1	10	10	-	1	1	-
recvfrom()	Y	Y	Y	1	2	2	2	1	1	1
setsockopt() (changed)	Y	Y	Y	2	4	8	8	-	-	-
setsockopt() (unchanged)	Y	Y	N	1	6	6	-	1	1	-
getsockopt() (changed)	Y	Y	Y	2	-	-	-	-	-	-
getsockopt() (unchanged)	Y	Y	N	1	1	1	-	1	1	-
select()	Y	Y	N	1	15	15	-	-	-	-
TOTAL					110	150	88 (→ 59%)	22	36	30 (→ 83%)

Table 1. List of the functions that had to be modified in FreeAmp and Gnucleus.

inserting dual-stack capabilities into the code. The code related to the session establishment has been modified to use some of the values obtained through the previous call to `getaddrinfo()`, thus avoiding hard-coded parameters. The code related to data reception has been modified only in case of UDP streams because the `recvfrom()` function needs a parameter containing the address of the host data are to be received from. This parameter has to be changed because the old `sockaddr` structure is not large enough for IPv6 addresses and it has to be replaced by a `sockaddr_storage` one. This problem does not exist for TCP-based streams because the function `recv()`, used in that case, does not require this parameter.

B. *Quantifying the effort*

In order to estimate the quantity of code that needs to be changed to convert an existing IPv4 application to a dual-stack application, we considered the number of socket calls in source files. This aimed at determining the portion of the socket interface that does not support IPv6 and, consequently, needs to be changed. However, some functions imply writing bigger quantity of code than others. For example, resolver functions (i.e. name to address and related) like `gethostbyname()` or `gethostbyaddr()` imply filling out socket address structures that are different in IPv4 and IPv6. The criterion chosen to quantify the entity of the change is "lines of code per function", i.e. the amount of lines of code that need to be modified for each function, which is explained as follows:

- Resolver functions: the change is considered being approximately as 4 lines of code for each function because we have to change both the function call and the code before it (for filling out the socket address structure).
- `connect()` and `bind()` functions: like resolver functions, they require socket address structures. However, often these structures have already been filled by the previous step (resolving an address happens before connecting or binding a socket) and only some additional manipulation is required. For this reason they have been considered being 2 lines of code.
- `getsockopt()` and `setsockopt()` functions: they are used to get (and set) flags that modify the behavior of the socket. Some of these options are related to the transport layer (e.g. TCP options), while others deal with network-level issues (e.g. join a specific multicast group). While the former do not need to be changed in IPv6, the latter uses different option codes in IPv4 and IPv6, therefore the code must be duplicated. These functions require a "lines of code factor" equal to 2.
- The entity of change for the rest of the socket functions is considered one line of code since no particular modification is required before calling them.

The result of the profiling can be seen in Table 1: in FreeAmp 88 lines of code (corresponding to the 59% of the lines of code related to the socket interface) have been changed; in Gnucleus, we had to change 30 lines of code (corresponding to the 83% of the lines of code related to the socket interface).

C. *Additional problems*

The previous section does not consider the amount of work required to localize the code that needs to be changed. This is relatively easy in case of "pure" socket functions (a "keyword search" often suffices), although the proper engineering of the code (i.e. strong modular organization) helps considerably. For instance, the Gnucleus code was definitely better than the FreeAmp one: an insight is given by the number of lines of codes related to the socket interface, which are 150 in FreeAmp and 30 in Gnucleus.

Furthermore, there are a couple of problems are related to the management of both IPv4 and IPv6 connections at the same time and that are not included in previous numbers. For instance, a client may try to contact a server (which has both IPv4 and IPv6 addresses) through the IPv6 address. If the application on the server host is able to accept only IPv4 connections (despite the IPv6 support within the server's operating system), the connection will be dropped after a timeout. To avoid loss of connectivity, additional code must be inserted into the client that forces it to connect through the IPv4 address as well (*fallback* mechanism)[1]. A similar issue exists for server applications as well: the application must be able to accept sessions on both IPv4 and IPv6 sockets, which means that the waiting socket (and the corresponding waiting thread) must usually be duplicated for every address family.

Unfortunately, there are additional parts of code that need to be changed. For instance, the code related to a custom graphical control that is used to insert an IP address must be changed. Perhaps, the most critical point is related to the code that manages the "presentation" of network addresses. For instance, the code that parses an input string (for example an URL) must take into account that the input string can be a literal name (e.g. "`foo.bar.com`"), an IPv4 address (e.g. "`192.168.0.1`") or an IPv6 address (e.g. "`fe80::2`"). In this case, the problem gets worse when also the "port" identifier can be specified, since the most common separator between address and port is the "`:`" sign in IPv4 (e.g. "`192.168.0.1:80`"), which is instead a intra-address separator for IPv6. Additionally, all the code that "prints" addresses (e.g. logging, printing) must be modified in order to support the new address format.

Applications that define a custom protocol for transferring data on the network deserve additional attention. For instance, Gnucleus exchanges the IP addresses of its peers between different machines. Obviously the format of Gnucleus packets must be changed in order to carry IPv6 addresses as well, but this could have serious consequences due to legacy compatibility. In other words, if we design a new (and simple) packet format, we incur into interoperability problems with older applications because we need to define a new "version" of the protocol. However, maintaining the compatibility could trigger the definition of a very elaborate packet format, which is preferable to avoid.

It can be concluded that the most part of the socket interface has to be changed in order to add dual-stack capabilities to an application. Unfortunately, this is only the simplest part of the job: the amount of code that needs to be changed outside standard socket API usually depends on the application and it can range from almost nothing to a considerable amount. The problem is that this code is very difficult to locate because a "keyword search" cannot be used in this case.

D. *IPv6-only applications*

An alternative approach to dual-stack applications consists in creating IPv6-only applications. This is much simpler because there is no requirement of being address-independent; therefore the programmer can simply use IPv6-specific structures instead of the IPv4 specific ones originally part of the code. However, this approach cannot avoid the most critical problems, i.e. the ones presented in Section II.C, which must be addressed explicitly.

However, the choice of creating an IPv6-only application only because it has lower complexity is a shortsighted approach; only dual-stack applications guarantee a smooth transition to the new network protocol.

E. *Cross-platform compatibility*

Cross-platform compatibility problems arise when using the new (i.e. the one devised for the deployment of IPv6) socket interface. The most significant differences are between Win32 (namely, the socket implementation in Windows XP Service Pack 1) and Unix. For instance, some of the functions aimed managing the list of the installed network adapters, `if_nameindex()`, `if_nametoindex()` and `if_indextoname()` are not available on Windows, although they can be emulated by means of platform-specific functions. Other examples include `inet_ntop()` and `inet_pton()`, which are used to translate an address from binary to "presentation" form and vice-versa, and are not available because the same result can be obtained by using the `getaddrinfo()` function with specific flags. Additional differences – e.g. in Win32 closing a socket requires `closesocket()` instead of `close()`, a socket descriptor is a `unsigned integer` instead of a signed one, the socket library must be initialized by means of the `WSAStartup()` and cleaned with the `WSACleanup()` – do not make writing cross-platform code easier.

Although far less significant, several differences exist between different Unix flavors as well. For instance, FreeBSD allows a single server socket to accept both IPv4 and IPv6 connections[2], while most of the other systems require two sockets. Linux by default generates a SIGPIPE signal whenever an error occurs on sending data on stream-oriented sockets, while other don't. Other examples may follow.

In conclusion, cross-platform compatibility is still an open issue even using the updates proposed by RFC 3493 in the socket interface.

III. Applications With Source Code Available: Special Cases

This section presents special cases that can occur in applications with source code available.

A. *Migrating applications based on external libraries with network-related functionalities*

Java or .NET-based applications provide a typical case of

[1] The IPv4-mapped address, defined in [10] allows connecting to an IPv4 host through an IPv6 address (although it can be disabled by means of the IPV6_V6ONLY socket option). However, this does not solve the problem and the fallback mechanism is always needed.

[2] Indeed, this is very useful in order to create a program that accepts both IPv4 and IPv6 connections seamlessly. Vice-versa, a program that wants to accept both IPv4 and IPv6 connections must open two distinct server sockets on other operating systems.

applications based on external libraries, where the underlying framework usually provides network-related functions. Applications based on Remote Procedure Calls (RPC) provide another example.

When network related functions are provided by an underlying framework, the porting is easier because the application will become almost IPv6-compatible as soon as its underlying framework becomes IPv6-compatible. Some problems may arise in case the application is to use new features, such as new API functionalities. In this case the task is much harder than before because each application has to detect the API version at execution time and use new features only if the run-time support makes them available.

The same considerations apply to applications that are based on some kind of general-purpose programming library (like Microsoft MFC, wxWindows, QT, etc.): the IPv6 support could be introduced almost transparently when the library becomes IPv6-compatible.

However, less apparent problems, particularly the ones highlighted in Section II.C (e.g., the input of network addresses and application-specific protocols), still remain and must be checked carefully because they are outside the responsibility of the framework the applications are based upon.

B. Migrating applications that detect IPv6 support at compile time

Several applications (mostly on UNIX) are distributed as a source code. The user downloads the source tarball, compiles it, and launches the application. However, from the application perspective this adds a new problem: does the operating system (on which the compilation occurs) support IPv6? The application must be provided with both old (BSD-style) and new (RFC 3493-style) code, and the most appropriate one must be selected at compilation time. Some tools (the GNU `autoconf` and `automake`) help in handling this issue; they define specific flags that will be used by the compiler to enable the correct version of the source code (usually through `#define` primitives). However, some of the work must still be done by hand, for example because the IPv6 support may be present, but limited to an old draft (e.g. it may support only `sockaddr_in6` structures instead of the more general `sockaddr_storage`).

IV. MIGRATING APPLICATIONS FOR WHICH SOURCE CODE IS NOT AVAILABLE

Given the large number of operating systems (and the growing number of applications) that support IPv6 and the benefits stemming from the deployment of IPv6, system administrators should introduce support for this protocol in their networks and favor native IPv6 access to their servers. However, rarely end users have the source code of their applications, and often applications are not (yet) ready for IPv6. This section presents the most important techniques that are currently available to "patch" old applications in order to make them IPv6-compatible without needing to manipulate their source code. This is considered particularly significant in the effort of porting server applications.

A. Network scenario

While there is no conceptual difference between modifying the source code of a client application versus a server application, different methods must be used when the source code is not available.

The most common scenario in the next future is depicted in Figure 3. A network site (e.g. a Company intranet) has IPv4 and IPv6 support, some of the clients have IPv6-capable operating systems and some applications are already able to exploit an IPv6 transport. The most impelling requirement for a network administrator is to update the server infrastructure (web servers, mail servers, etc.) in order to permit native access to IPv6 clients, leaving the porting of clients' applications as a next step.

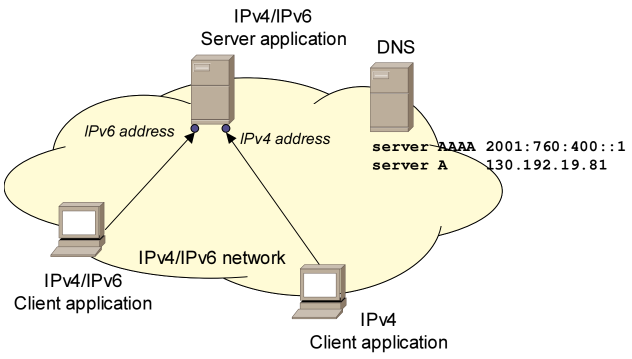

Figure 3. The most common IPv6 deployment scenario in the next future.

The work presented in this paper assumes that all the servers can be converted to dual-stack (i.e. both IPv4 and IPv6 protocol stacks are installed on the machine), which is a correct assumption for most cases. Vice-versa, clients can be either IPv4-only or dual stack. For instance, dual-stack hosts are considered the best solution at least in the short-medium period. Dual-stack servers may run both IPv6-compatible applications (such as the well-known Apache web server) and IPv4-only ones, with a mix of them installed on the same machine (e.g. an IPv6-compatible secure shell daemon and an IPv4-only mail server). Each application will use its preferred transport.

The following assumes that the site's DNS server is able to handle IPv6 addresses (`AAAA` records) and contains both an IPv4 and an IPv6 mapping for each entry related to dual-stack servers (e.g. `myserver.mydomain.com`). Notice that the DNS service is not required to answer to queries by means of an IPv6 transport.

B. Migrating server applications

This section explains the methods to make the server infrastructure IPv6-compatibile from the applications point of view.

1) *TCP/UDP Port Forwarders*

This is one of the preferred ways to make an IPv4-only server application capable of interacting through an IPv6 transport. The server machine needs a TCP/UDP Port Forwarder (often called *Bouncer*), that is an (IPv6) server application waiting for TCP/UDP connections to an IPv6 address and a specific port. The structure of this tool is shown in Figure 4. The Port Forwarder includes a server module that opens a server socket waiting for connections on the IPv6 address of the dual-stack server using the same transport protocol (TCP/UDP) and port of the IPv4 server application. When a new connection is accepted, a client module establishes a new connection[3] (inside the server) toward the IPv4 application. The IPv4 server application will simply see a new connection arriving from its own — `localhost` (127.0.0.1) — address. The communication path from the client to the server is split into two connections: an IPv6 one and an IPv4 one. The Port Forwarder service will forward all the payload data received from the client on the IPv6 connection to the server on the IPv4 one, and vice versa.

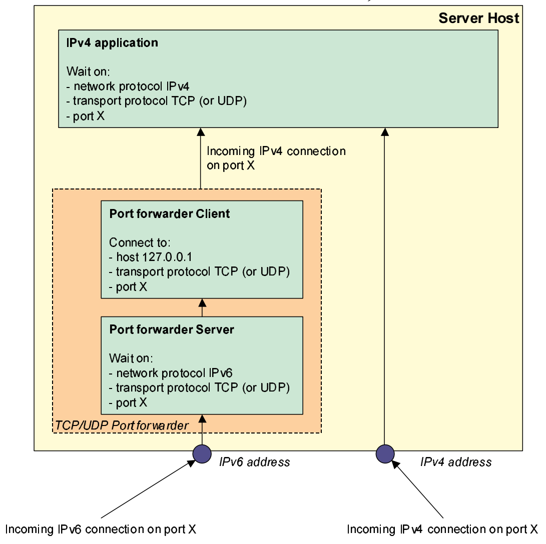

Figure 4. Running a TCP/UDP Port Forwarder.

This mechanism has several practical advantages. First, the intrusiveness is limited to a small server that waits on a specific port. This allows the coexistence of IPv4-only applications (which will use the port forwarding service) and IPv6-capable ones (which will accept directly IPv6 connections) on the same machine. Second, the overhead of this solution is fairly limited thanks to the fact that the Port Forwarder does not change the incoming data. Third, it works with most of the TCP/UDP applications because it does not get involved with the data payload (which is application-dependent).

Its most important limitations are related to some applications (like FTP servers in "standard mode") in which the server opens a new connection toward the client: the Port Forwarder is unidirectional with respect to the incoming connections, which must arrive from outside. Other limitations are related to applications that include network addresses in their payload. For instance, a web server with virtual hosts (i.e. a web server that returns different pages according to the host name contained into the requested URL) is problematic because both its IPv4 and IPv6 addresses must share the same host name into the DNS database, otherwise the URL changes and the server cannot understand which virtual host we are referring to. Another limitation is due to the masquerading of the real address of an incoming connection (all the packets received by the application are coming by the `localhost` address), which can be useful particularly for statistics, accounting and filtering (e.g. access lists). In case some of these features are needed, the Port Forwarder must include a statistics module taking care of the necessary accounting when a session is accepted on it. Finally, some problems are related to packet fragmentation, which arise particularly with UDP traffic (e.g. multimedia sources that use the maximum MTU length). Packet fragmentation may be required to fit a message coming from the IPv4 application into an IPv6 packet due to the larger IPv6 header size.

Figure 5 shows a typical deployment scenario of this mechanism. Although this is the preferred configuration, the Port Forwarder service can be alternatively installed on a third machine as well. In that case, it will forward traffic within IPv4 packets on the network toward the destination server machine.

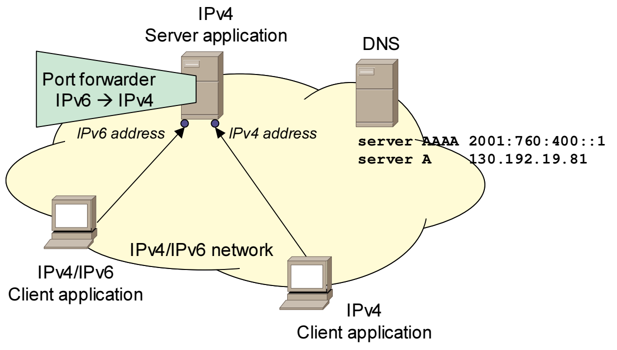

Figure 5. A TCP/UDP Port Forwarder example of application migration.

A possible enhancement of this mechanism consists in making the destination IPv4 address, used to open the second connection, dynamic according to the idea of the RFC 3142 (An IPv6-to-IPv4 Transport Relay Translator) [4]. In this proposal the client uses a special IPv6 address that embeds the IPv4 one in the last 32 bits. A set of Port Forwarders servers can be deployed throughout the network and will serve all the "real" servers within a specific domain. This solution has lower configuration overhead with respect to the traditional Port Forwarder deployment (a statically configured IPv4

[3] While TCP actually establishes a connection between two machines, the term "connection" sounds inappropriate for UDP traffic. However, we can define a connection for UDP traffic as well, intending all the traffic that is exchanged between two hosts using the same source and destination ports. Since UDP does not have a mechanism to tear down a "connection", a timeout mechanism will be needed to "tear down" the connection when no traffic is present for a given amount of time.

address of the server to connect to is not needed). However, it has also a drawback: the Port Forwarder becomes a single point of failure. This problem does not exist in the original deployment because both a Port Forwarder and its server application reside on the same machine.

A well-known Port Forwarder is the 46Bouncer [5] tool, which is currently being used on several servers in our University network because of its efficiency and its cross-platform compatibility.

2) *TCP/UDP Relay*

The IPv6-to-IPv4 Transport Relay Translator [4] (RFC 3142) mimics the Port Forwarder; the difference is that the relay is not identified by means of a unique address. When and IPv6 host attempts to connect to a server that does not have an IPv6 address, the (modified) DNS returns an IPv6 address in the form `[64_bits_network_prefix]::A.B:C.D.`, where the network prefix is the one in which the TCP/UDP relay has been installed. RFC 3142 does not specify the details of how this mechanism is implemented: by means of a modified DNS resolver in the client application, or a modified DNS server, or other alternatives. The resulting IPv6 packet is routed toward the relay, which terminates the first connection and opens a new one toward the server.

Even though this mechanism is more advanced than the TCP/UDP Port Forwarder, they share the same principles. A TCP/UDP Relay may require less configuration effort than a Port Forwarder (the IPv6 address is derived from the IPv4 host); however the same machine must have several IPv6 addresses (one for each IPv4 machine it is going to serve) and it may become a single point of failure (in the most common scenario several servers share the same relay).

3) *NAT – PT*

The Network Address Translator – Protocol Translator (NAT-PT) [2] extends the traditional NAT paradigm with protocol translation, i.e. it transforms an IPv4 packet into an IPv6 one and vice-versa. Figure 6 shows a possible deployment scenario: a NAT-PT is used to transform IPv6 packets coming from a client into IPv4 packets to an IPv4-only server.

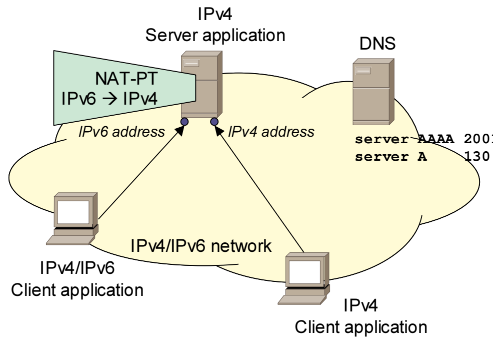

Figure 6. A NAT-PT example for application migration.

This technique provides transparent connectivity from IPv6 end-nodes to IPv4 only server applications. This is achieved using a combination of Protocol Translation based on SIIT (Stateless IP/ICMP Translation Algorithm) [3] with the dynamic address translation of NAT and appropriate ALG (Application Level Gateway).

The scenario shown in Figure 6 is not the typical NAT-PT deployment scenario. In general, NAT-PT provides connectivity between an IPv4-only network and an IPv6-only one, which is not our goal. Figure 6 proposes an adaptation of the NAT-PT mechanisms for a simpler objective, i.e. translating IPv6 packets that are arriving at a server into IPv4 packets, which can be delivered to an application running within the server itself. The NAT-PT mechanism does not perfectly suit our purposes since it will translate all IPv6 incoming packets, preventing IPv6-capable applications on the server host from receiving IPv6 packets.

Besides all the other NAT-PT known problems (e.g., it does not allow to start a connection from the host on which it is deployed), the above limitation prevents NAT-PT from being a valid solution to the problem of porting IPv4 applications running on dual-stack servers. NAT-PT is more appropriate in case of an IPv4-only machine communicating through an IPv6-only network.

4) *Bump-In-the-Stack*

The Bump-In-the-Stack (BIS) [6] technique can be seen an evolution of a NAT-PT service, and it is embedded into an host. The BIS is installed on server hosts and acts basically as a sophisticated address translator.

In the case a server receiving a connection, the behavior resembles the NAT-PT one: incoming IPv4 packets are transferred directly to the application, while incoming IPv6 packets are transformed into IPv4 ones and then transferred to the application. In the case a server initiating a connection, the required steps are more elaborate. The application usually begins by sending a DNS query looking for an `A` record. The BIS module intercepts the query and sends two queries out of the box, one for an `A` (i.e. an IPv4 address) record and the other for an `AAAA` (i.e. an IPv6 address) one. If the DNS reply contains only an IPv4 address, the application starts exchanging IPv4 packets. On the other hand, if the reply reports an IPv6 address, the BIS module maps it onto a local IPv4 address (taken from a locally defined pool) that is returned to the IPv4 application. Hence, the application will start sending packets to such IPv4 address, which are intercepted by the BIS module, translated into IPv6 packets, and sent out to the destination.

This technique has the advantage of allowing the server application to initiate a connection, but it suffers from the same limitation of the NAT-PT approach: all incoming IPv6 packets are translated into IPv4 ones. Therefore the server host cannot run IPv6 native applications unless it has two IPv6 addresses: one to be translated into IPv4, and one bound to an IPv6 native stack.

5) Bump-In-the-API

The Bump-In-the-API (BIA) [7] technique is an evolution of the previous one in the sense that the implementation of the socket interface supports old IPv4 structure and functions, but can generate IPv6 packets. The behavior is almost the same of BIS, although no packet translation is performed: the modified socket implementation is able to distinguish if the communication has to be done toward an IPv4 address or toward an IPv6 one and generates the packets accordingly. In the same way, upon receiving an IPv6 packet, the socket implementation checks if the server waiting on the destination port is IPv4 or IPv6 and delivers the data appropriately.

The BIA technique is highly efficient and does not suffer from most of the BIS limitations. However there is only one implementation (provided by ETRI, Korea) for Windows 2000 of this mechanism at the time of writing, which seems to be unavailable for general use.

6) SOCKS

The SOCKS-based IPv6/IPv4 gateway mechanism [8] is based on the SOCKS protocol (SOCKSv5 [9]), which is designed to provide a framework for client-server applications in both the TCP and UDP domains to conveniently and securely use the services of a network firewall. The protocol is conceptually a "shim-layer" between the application layer and the transport layer. This proposal is based on a mechanism that creates two connections (one between the application and a SOCKS server, and another between the SOCKS server and the other party) in a way that looks similar to the port forwarding mechanism. However, this proposal has been designed with client issues in mind (i.e. an application that wants to "connect" to a server) instead of server issues (i.e. an application meant to "accept" incoming connections). It follows that this mechanism is not the best choice for server applications.

7) Application-Level gateways

Application-Level Gateways are able to decouple clients and servers since they act like a server toward the client, and like a client toward the server. A gateway understands the semantics of the data exchanged by client and server (and usually modifies it), while a port forwarder is a transparent box.

There are several implementations of application-level gateways (web proxies are the most common ones) in the IPv4 world and we cannot see any particular problem in creating an application–level gateway that accepts IPv6 connections from clients and connects to servers by means of IPv4. The biggest problem of this mechanism is that a distinct gateway is required for each application, while the port forwarder is application-independent. The advantage is that an application level gateway works also with peculiar applications, e.g. the ones in which the server opens a connection toward the client.

C. Migrating client applications

Most of the techniques presented in the previous section can be used to add a partial IPv6 support to IPv4-only applications, both client and servers. However, clients are usually opening a connection toward the server, which makes the deployment scenario different from the one related to server applications and critical for some of the presented techniques.

For instance, the TCP/UDP Port Forwarder cannot be applied to client applications because it statically defines the address of the host the connection should be terminated to. This approach works for applications that are going to connect always to the same server (like mail clients), but it is not appropriate for other applications that usually open a connection to different servers (like web browsers). For instance, an approach based on TCP/UDP Relay is more appropriate.

While almost all the techniques (but the TCP/UDP port forwarder) can be used for client-side migration, the Author does not believe this is a hot topic at present time; furthermore experience is still missing. An in-depth analysis of these issues is left for future work.

V. A CASE STUDY: THE POLITECNICO DI TORINO NETWORK

The support for IPv6 has been introduced in the network infrastructure of out University several years ago and only a few secondary sites remain without IPv6 connectivity.

Unfortunately, almost all our server applications do not support IPv6, with the exception of DNS servers (which operate with Bind 9). Other services (mail, web, a Shoutcast server[4]) are being offered over IPv6 transport trough a TCP/UDP Port Forwarder installed on IPv6-enabled operating systems (Linux, Windows 2000 and Compaq Tru64). Finally, a double entry has been created in the DNS such that two addresses (one IPv4 and one IPv6) correspond to the same name. Services running on an old server that does not have an IPv6 stack are being provided over IPv6 transport by means of a TCP/UDP Port Forwarder installed on another machine.

The results are interesting: IPv6 clients are able to use IPv6 to connect to vast majority of the services. Although the outcome is highly positive (mostly because of the simplicity and low intrusiveness of the Port Forwarder mechanism, which can be enabled on a per-TCP/UDP port basis), some problems have been noticed. A first set of issues stems from the fact that any connection is originated from the same IPv4 address from the point of view of the server application. Besides preventing the server application from keeping statistics about IPv6 connections, some more serious issues are related to access lists. For instance, our SMTP server is configured to forward only messages originated from local IPv4 addresses; unfortunately, all IPv6 connections appear to be local. To solve this issue (which is essential to avoid spam) the Port Forwarder had to be configured to accept connections coming only from IPv6 hosts on our network.

Another issue is related to IPv6 clients trying to connect to

[4] A Shoutcast server is a multimedia server that broadcasts audio through a TCP transport.

a non-existing IPv6 service. For example, in one instance a Port Forwarder was by mistake not activated on a port used to manage a web server. The result was an unexpected delay when opening a connection from a dual-stack client to the management port, because the client sent an IPv6 request (that never got an answer) and it switched to IPv4 only after a timeout.

Finally, one issue arose for the old server that does not have an IPv6 stack: when trying to connect to the `ssh` service through IPv6, the connection was terminated on the "port forwarder" host. The reason is that the Port Forwarder itself runs an `ssh` server. Consequently, an IPv4 address is to be used when connecting to some services on the old IPv4 only server that are also active on the host on which the port forwarder is installed. With respect to this point, the Port Forwarder runs better when the IPv4-only application and the Port Forwarder are installed on the same machine.

The port forwarder was successfully deployed for UDP traffic as well. An MP3 jukebox has been installed that is sending IPv4 multicast traffic; this is translated into IPv6 multicast. An old DNS server (which supports `AAAA` records but not IPv6 transport) was made IPv6-compatible by means of a Port Forwarder.

VI. Conclusions

This paper includes two main contributions. The first one consists in providing a description and quantitative evaluation of the effort required to add IPv6 support into an application's source code. Although the number of applications we modified is fairly small, and therefore our results have limited validity, our findings show that such effort cannot be considered negligible. Preliminary results show that far more than 50% of the lines of code related to the network interface must be modified. Furthermore, some additional parts of the application (e.g. custom network protocols, input forms, address "presentation" issues, URL parsing), less obviously requiring changes and present in applications at largely varying levels, can significantly impact the cost of the task.

The second contribution relates to the problem of providing native IPv6 services (web, mail) when the source code of server applications is not available. Among all the methods that have been presented, the most important one are the TCP/UDP Port Forwarding and Bump-In-the-API techniques. The former is very simple and a few implementations are widely available, while the latter is more complete, but more complicated and there are no widely available implementations.

Finally, a case study of our University information system has been presented showing that under some constraints (IPv4/IPv6 network and mainly dual stack servers) the most important network services have been successfully converted to IPv6 by means of the TCP/UDP Port Forwarding mechanism. Some issues arose, but the simplicity, the low intrusiveness (and the availability) of the Port Forwarder made it a better solution than other more sophisticated techniques.

Acknowledgment

The author thanks Ernesto Escalona for its help in adding IPv6 support in FreeAmp and Gnucleus during his graduation thesis. He would like to thank also Lorenzo Colitti and Mauro Tortonesi for their several suggestions. Finally, a special thank goes to Mario Baldi for its several suggestions and its invaluable contribution in reviewing (and cleaning) this paper.

The work presented in this paper has been partially founded by Telecom Italia Labs, Torino (Italy) and by the European Commission under the 6Net project.

References

[1] R. Gilligan, S. Thomson, J. Bound, J. McCann, W. Stevens, "Basic Socket Interface Extensions for IPv6", *RFC 3493*, Internet Engineering Task Force, February 2003.

[2] G. Tsirtsis, P. Srisuresh, "Network Address Translation - Protocol Translation (NAT-PT)", *RFC 2766*, Internet Engineering Task Force, February 2000.

[3] E. Nordmark, "Stateless IP/ICMP Translation Algorithm (SIIT)", *RFC 2765*, Internet Engineering Task Force, February 2000.

[4] J. Hagino, K. Yamamoto, "An IPv6-to-IPv4 Transport Relay Translator", *RFC 3142*, Internet Engineering Task Force, June 2001.

[5] F. Risso, *46Bouncer*, Politecnico di Torino, September 2001. Available for download at http://netgroup.polito.it/46Bouncer/.

[6] K. Tsuchiya, H. Higuchi, Y. Atarashi, "Dual Stack Hosts using the "Bump-In-the-Stack" Technique (BIS)", *RFC 2767*, Internet Engineering Task Force, February 2000.

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

[8] H. Kitamura, "A SOCKS-based IPv6/IPv4 Gateway Mechanism", *RFC 3089*, April 2001.

[9] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, L. Jones, "SOCKS Protocol Version 5", *RFC 1928*, March 1996.

[10] R. Hinden, S. Deering, "IP Version 6 Addressing Architecture", *RFC 3513*, Apr 2003.