Protocol Independence in Applications for Network with Different IP Versions

João Cândido Dovicchi, *IEEE Member*, and João Bosco da Mota Alves

***Abstract*—This paper describes some strategies to port network Application Program Interfaces (API) to support both types of Internet Protocol (IP). Authors strongly suggest that coding program for dual protocol support should be done as protocol independent as possible. Some hints on how to deal with sockets to avoid excess of #ifdef instructions in program porting code is discussed, and the substitution of explicit name resolver functions (gethostbyname and gethostbyaddr) for POSIX-defined getaddrinfo is also commented. Some issues about Domain Name Services (DNS) is also presented.**

I. INTRODUCTION

NEW version 6 of Internet protocol (IPv6) is the next generation standard for network communication, described by Internet Engineering Task Force (IETF) [1]. It increases the IP address size from 32 bits to 128 bits, to support new features for Application Program Interface (API) as, for example:

- Larger and flexible address capability
- Efficiency in packet forwarding and delivery
- Native support for secure communication and differentiated services
- Enhanced mobility support and ease of management

IPv6 has native support for multicast, unicast addresses [2] and future implementation of anycast address capability [3].

Brazilian high speed network, as known as RNP2, is being implemented and should be fully working in less then one year [4]. As already pointed, the backbone of the new Brazilian network will take some time, not to say some years, to completely migrate from an Internet Protocol, version 4, (IPv4) to IPv6. This means that in this mixed network there still be IPv4 and IPv6 packets to be routed (see Fig 1). In previous work we have suggested some hints on how encapsulated IPv6 on an IPv4 packet could be routed over a mixed network to allow client/server applications to communicate with improved Quality of Service (QoS) [5].

Some important spots about IPv4 and IPv6 are:

- Multicasting support is optional in IPv4 but mandatory in IPv6.
- IPv6 has no support for broadcast. This means that any IPv4 application that uses broadcasting must be transcribed and coded for IPv6 multicasting.

J. Dovicchi is professor at Uberlândia Federal University and researcher at the Advanced Centre for Computing on Sonic and Multimedia (NACSM) of the Uberlândia Federal University (UFU), Uberlândia, MG, Brasil (email: dovicchi@ieee.org)

J. B. M. Alves is professor at Santa Catarina Federal University and researcher at the Remote Experimentation Laboratory (RExLab) of the Santa Catarina Federal University, Florianópolis, SC, Brazil (email: jbosco@inf.ufsc.br)

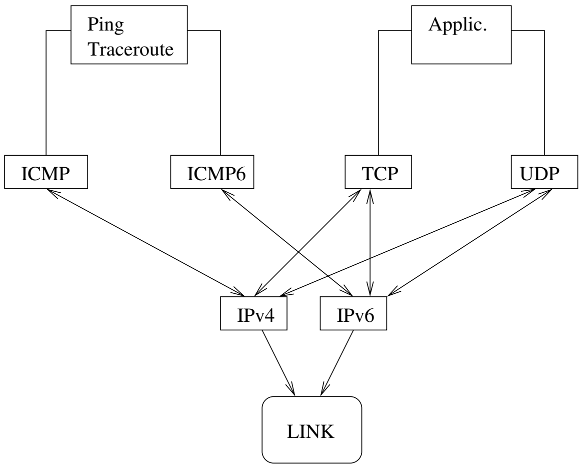

Fig. 1. Mixed protocols network

- Broadcasting and multicasting require UDP and do not work with TCP.

Routing IP packets over a mixed network environment demands a fine control of configurations, once these protocol types are fundamentally different, and there are specific rules for routing broadcast, multicast and unicast packets.

There are 4 types of connection to consider in mixed network architectures:

1) IPv4/IPv4 - Broadcast (UDP, ICMP, IGMP) and unicast (TCP)
2) IPv4/Dual-stack - Broadcast (UDP, ICMP, IGMP), multicast (UDP), unicast (TCP)
3) IPv6/Dual-stack - Multicast (UDP, ICMP6, IGMP6), unicast (TCP)
4) IPv6/IPv6 - Multicast (UDP, ICMP6, IGMP6), unicast (TCP)

In order to guarantee client/server connections between IPv4 and IPv6 hosts, it is necessary to map IPv6-to-IPv4 and, communication of IPv6 hosts or networks through an IPv4 segment demands tunneling the connection (see table I).

Even though some network segments will be operating with IPv6, Brazilian National Research Network (RNP) will have to support both IP types traffic. Obviously there will be dual-stack servers and clients as well as gateways to tunnel IPv6 over IPv4 segments. Figure 2 shows dual-stack server accepting IPv4 and IPv6 connections and figure 3 shows IPv6 encapsulation inside IPv4 datagrams for tunneling over IPv4 network.

Since multimedia real time applications use packets stream, another problem is the necessity to control bandwidth resource reservation, which can be done with *Resource Reservation Pro-*

Client	Route	Server	Map IPv4/IPv6	Tunnel
IPv4	IPv4	IPv4	no	no
IPv4	IPv4 IPv6	IPv6 dual	yes	no
IPv6 dual	IPv4 IPv6	IPv4	yes	no
IPv6	IPv4 IPv6	IPv6	no	no
IPv6	IPv4	IPv6	yes	yes

TABLE I
IP MAPPING AND TUNNELING OVER MIXED NETWORK

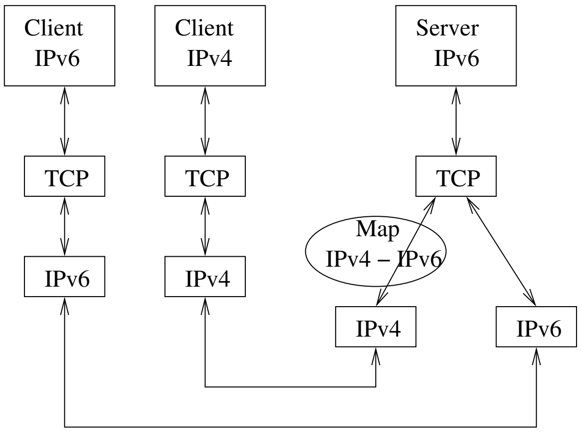

Fig. 2. IPv6 server, runing dual-stack protocol answering IPv4 and IPv6 requests

tocol (RSVP) that is more reliable to implement QoS than *Real Time Protocol* (RTP). Once RTP is UDP based, there is no way to implement it with QoS and, as IPv6 has field descriptor for flow control, RSVP can be implemented to control bandwidth and guarantee QoS [6]. However, implementation of bandwidth control on tunneling IPv6 over IPv4 segments has to be done in the API code level.

Besides, in this new scenario, we have to consider that Transmission Control Protocol (TCP) is used only for unicast addresses while User Datagram Protocol (UDP) supports other paradigms and TCP is more reliable than UDP. Even though, during last five years, many network architectures to manage IP differences have been proposed, all of them are based on three paradigms: dual-stack hosts/clients; tunneling; and IP translation (or encapsulation). An important review of these IETF documents was recently published [7].

In this paper, authors show some basic problems on porting IPv4 API to support both types of protocols, point to some hints on helping portability for programs that use socket functions to communicate over IP, and discuss them with focus on new paradigms of socket programming.

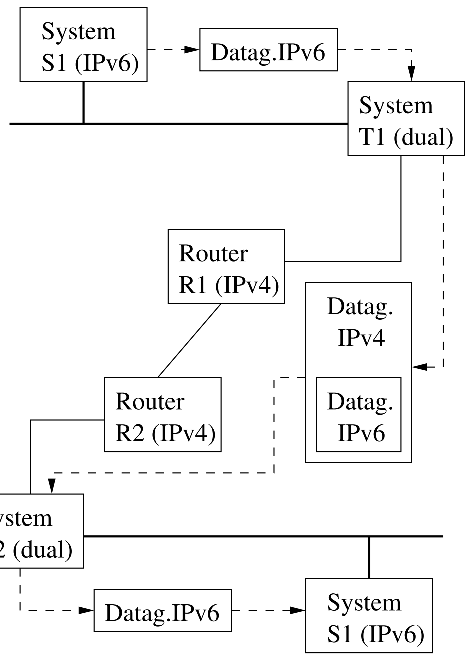

Fig. 3. IPv6 encapsulated on IPv4 datagrams for tunneling over a mixed network with an IPv4 routing segment

II. THE MAIN PROBLEM: PORTABILITY

Although some new network applications are coded for IPv4/IPv6 interoperability, there are many of them that still support only IPv4. In the case of programs that depend heavily on IPv4 functions, such as multicasting, IP options and raw sockets, recoding for IPv6 or dual-stack support can be much more difficult.

API porting for IPv4 and IPv6 support depends on types of reliability of the protocol to be used. This is mainly a choice between TCP or UDP applications. Someone can argue that UDP usage will decrease in next generation Internet in comparison to TCP connections. This argument is based on the fact that new TCP implementations and today's network are faster than they were in the past. Besides that, the increasing bandwidth resources of Internet2 could point into direction that software designers should recode UDP applications to TCP paradigm.

However, the increase of real-time multimedia applications (video conference, IP telephony, real-time audio and video delivery etc.), which implies multicasting will increse also UDP usage. Although there must be considered that UDP is less reliable than TCP, advanced UDP sockets programming can add reliability to UDP [9].

The procedure of coding an application with IPv4 and IPv6 support has to deal with the problem of knowing if the end-user system has or not support for IPv6. So, source code has to be compiled within a correct environment. The first thing that occurs to a programmer, when planing an API port, is to trap IP type configuration for that system during compilation time using `#ifdef / #else / #endif` structure. Once IPv6 clients can communicate to IPv4 servers, and vice versa, conditional expressions are an obvious choice. Lets consider, for example, a case where the code has to decide which socket to use:

```
...

#ifdef CONFIG_IPV6
  int                 af;
  struct sockaddr_in6  client;

  af = AF_INET6;
  Setsockopt (STDIN_FILENO,\
              IPPROTO_IPV6,\
              IPV6_ADDRFORM,\
              &af, sizeof(af));
#else
  int                 listenfd, connectfd;
  struc sockaddr_in    server, client;

  listenfd = Socket (AF_INET, SOCK_STREAM, 0);
  Bind (listenfd, &server, sizeof(server));
  Listen (listenfd, LISTENQ);
#endif

socklen_t       clilen;

clilen = sizeof(cli);

...
```

In this case, once defined the variable `CONFIG_IPV6` the code will be compiled with one socket type or with other type if the variable is not defined. However, other problems may arise from this solution. Some pieces of source code can become so littered with `#ifdefs` that it can be difficult to understand and maintain it.

One possible solution is to construct functions that are protocol independent. This can be done by redefinition of functions that has different structures in IPv4 and IPv6. One fine example is the difference between structures *sockaddr_in* and *sockaddr_in6*. The former returns a dotted-decimal IPv4 address and the ladder returns the hexadecimal string form of an IPv6 address and, since name conversion functions have to deal with *numeric* to *presentation* whith different data length for each IP version type. The resulting code is protocol dependent, for example:

```
...

static char str [128]

struct sockaddr_in {
   sa_family_t sin_family;           /* AF_INET */
   unsigned short int sin_port;
   struct in_addr sin_addr;
   unsigned char __pad[__SOCK_SIZE__ - \
     sizeof(short int) - \
     sizeof(unsigned short int) - \
     sizeof(struct in_addr)};
};

struct sockaddr_in6 {
   unsigned short int sin6_family;    /* AF_INET6 */
   __u16 sin6_port;
   __u32 sin6_flowinfo;
   struct in6_addr sin6_addr;
   __u32 sin6_scope_id;              /* RFC2553  */
};

#ifdef INET6_ADDRSTRLEN
   inet_ntop (sin6_family,
              &in6_addr.
              sin6_addr,
              str,
              sizeof(str));
#else
   inet_ntop (sin_family,
              &in_addr.
              sin_addr,
              str,
              sizeof(str));
#endif

...
```

The whole code could be reduced to one line if one writes a function that takes a pointer to a socket address structure, and calls the appropriate function to return the presentation format of the address. For example:

```
char *mysock_ntop (const struct sockaddr *sockaddr,
                   socketlen_t addrlen);
```

Where the structure *sockaddr*, which points to an address structure, and *socketlen_t*, that returns its length, can be defined in one header file. So the presentation format is returned as dotted IPv4 or hex IPv6 address, what makes it protocol independent. All `#ifdef` instructions can be better organized in a few libraries, making the code simpler.

One extra difficulty in code portability is inherent to the way IPv4 and IPv6 systems looks for domain names. Lets consider the situation in which the end-user system does not support IPv6 while local name server does. In this case, local name server will return both `IN AAAA` as well as `IN A` records of the peer the API is trying to connect. If the API is compiled to support both IP types, it calls `socket` to create an IPv6 socket type, and will fail to do so, since the system does not support IPv6.

To solve this problem, the API has to ignore the error from `socket` functions and try the next address from the name server record. If the name server is correctly configurated, it will return a `IN A` record for each `IN AAAA` and the opening of an IPv4 socket will succeed (see Fig. 4).

There are also two protocol dependent functions, namely *gethostbyname* and *gethostbyaddr*, that must be carefully used. When calling *gethostbyaddr* it is necessary to know which socket structure must be used (`struct sockaddr_in` or `struct sockaddr_in6`) and, of course, which member of the structure returns the binary address (`sin_addr` or `sin6_addr`). One example is in the *daemon.c* code of the sendmail program in the following excerpt:

```
# if NETINET
  if ((in_addr.s_addr = inet_addr(&name[1]))\
                              != INADDR_NONE)
                  hp = sm_gethostbyaddr((char *)&in_addr,
                                        INADDRSZ,AF_INET);
# endif
# if NETINET6
  if (hp == NULL &&
      inet_pton(AF_INET6, &name[1], &in6_addr) == 1)
      hp = sm_gethostbyaddr((char *)&in6_addr,
           IN6ADDRSZ,
           AF_INET6);
# endif

...
```

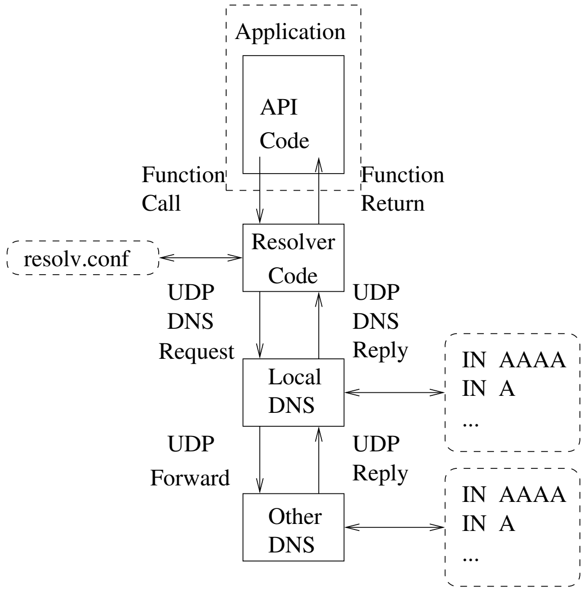

Fig. 4. DNS request for IPv4 and IPv6 clients

However, there is a POSIX standard [10] which describes protocol-independent interfaces and recommends use of *getaddrinfo()* that deals with name translation of a service location and/or a service name to set socket addresses. Some Unix™ OS define it in `netdb.h` include file:

```
extern int getaddrinfo (__const char *__restrict __name,
                        __const char *__restrict __service,
                    __const struct addrinfo *__restrict __req,
                 struct addrinfo **__restrict __pai) __THROW;
```

In this case, `getaddrinfo` is an abstraction that can replace `gethostbyaddr`. It is necessary to remember that its result is totally dependent on a fine tunning of DNS records.

III. Discussion

As stated previously, adding support for both IPv4 and IPv6 in the same source should grant not only the re-usability of the same code, but also must provide abstract structures for IP addresses and operations with them. This can allow multiple routing tables and route filters in the server program. So, defining a structural representation for `ip_addr` prevents it to be handled as a number and then can be used for *numeric to presentation* (ntop) function to return data in human readable format. A header file can describe each structure to be handled for IPv4 and IPv6 while the program uses the `ip_addr` function. For example:

```
...

#ifdef IPV6
    typedef struct ipv6_addr {
      u32 addr[4];
    } ip_addr;
#else
    typedef struct ipv4_addr {
      u32 addr;
    } ip_addr;
#endif

...
```

Considering OS dependence is another problem since IP stack differ from one OS to another and even between versions of the same OS some differences can be found. In this case, porting has to be done by changing or replacing isolated code in special modules or libraries.

It is evident that the great difference between IPv4 and IPv6 is their addressing capability. Connection over an ethernet network depends on Mediun Access Control (MAC) address binded to IP and host names of peers. It is determined by functions *gethostbyaddr* and *gethostbyname*. So the programmer has to be careful with DNS queries and answers. Even if an abstract function such as `getaddrinfo` is implemented, a precise DNS server configuration is mandatory for everything to work as expected. Besides this, MAC labels are addressed by 48 bits inside IPv4 header and 64 bits in IPv6. Then, porting API to IPv6 demands encapsulation of 48 bits of MAC address in 64 bits space[1]. This solution is already described [11].

Notice that if the looked-up domain service of a peer returns two IP addresses, one must take into account the order they are returned by the DNS. Many DNS servers have the bad habit of sorting the returned addresses, so it is not guaranteed that the answer will not be the perfect match for the query.

After that, there are some other situations one may consider when API ports are done. RFC documentation [1] recommends that IPv6 socket should be binded to IPv6 wildcard addresses. This was done to make IPv4 connections to IPv6 hosts possible. So, when an IPv4 packet is received it is mapped into an IPv6 address. For example, `192.168.1.1` would be mapped to `FFFF::192.168.1.1`. Of course, the intention was to make easy the migration from IPv4 application to IPv6, oppening only one socket on a given port for its API.

It is easy to predict that any firewall or access permission could block an IPv4 host or net, lets say `192.168.2.0/24`, but would let `FFFF::192.168.2.XXX` host through. Of course this is not a good idea. Authors strongly suggest that Berkeley Internet Name Domain (BIND) version 9 documentation should be read for a better comprehention of this problem and hints on how to solve it.

IV. Conclusions

Authors know that it is impossible to cover all problems linked to migration from IPv4 to IPv6. As they point in this paper, show that porting API is not a trivial process and, far from being totally exausted, they give some hints to help dealing with the problem.

Brazilian National Research Network (RNP) will migrate gradually from an IPv4 to IPv6 based backbone (RNP2) and, API design for supporting IPv4 and IPv6 must be considered with care.

[1] Recall that even modern ethernet devices support only 48 bits of MAC address.

During making of it, programmers must take into consideration some important notices:

- Portability - must attend OS differences and differences between different versions of the same OS.
- Flexibility - must be as flexible as possible considering different network architectures, network IP types segments and routers specifications.
- Readability - codes have to be modular, and structured into libraries to ease maintainance and recoding.
- Security - socket functions abstraction must antecipate security faults and holes for possible security failures.

As anyone can see, today's Brazilian backbone is a DNS-messy network. It is not difficult to find servers with absence of `IN A` records for hosts or domains and, even worse, lacking of `IN PTR` reverse records for hosts. It is very important to recall that good DNS servers configuration is mandatory for a mixed IP versions backbone to work.

REFERENCES

[1] R. Hinden, and S. Deering, "IP Version 6 Addressing Architecture," *IETF - Network Working Group*, RFC 2373, Jul. 1998.

[2] S. Deering, and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," *IETF - Network Working Group*, RFC 2460, Dec. 1998.

[3] C. Partridge, T. Mendes, and W. Milken, "Host Anycasting Service," *IETF - Network Working Group*, RFC 1546, Nov. 1993

[4] RNP "RNP2 - Infra-estrutura Internet2 para o Brasil,", Rede Nacional de Pesquisa, 2002. http://www.rnp.br/rnp2

[5] J. C. Dovicchi, "As Arquiteturas Cliente/Servidor na Nova Sociedade da Informação, usando IPv4 e IPv6 em uma rede de protocolos heterogêneos," *Anais do II TecEAD* - II Seminário Nacional de Tecnologia para EAD, Universidade Federal de Uberlândia, Uberlândia, MG, Brazil, Published in CDROM, Jun. 2002. http://www.ead.ufu.br/tecead_II

[6] S. Herzog, "RSVP Extensions for Policy Control," *IETF - Network Working Group*, RFC 2750, Jan. 2000.

[7] D. G. Waddington, and F. Chang, "Realizing the Transition to IPv6," *IEEE Communications Magazine*, Vol. 40 (6), pp. 138-148, Jun. 2002.

[8] R. Gilligan, and E. Nordmark, "Transition Mechanisms for IPv6 Hosts and Routers," *IETF - Network Working Group*, RFC 2983, Aug. 2000.

[9] W. R. Stevens, *Unix Network Programming*, 2nd. Edition, Upper Saddle River:Prentice Hall, ISBN 0-13-490012-X, 1998.

[10] IEEE "Standard for Information Technology - POSIX(R) Ada Language Interfaces- Part 1: Binding for System Application Program Interface (API) - Amendment 2: Protocol-Independent Interfaces," *IEEE Computer Society*, Portable Applications Standards Committee, IEEE Standards Board, Approved 8 December 1998.

[11] IEEE "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority," *IEEE Computer Society*, IEEE Standards Board, 1997. http://www.standards.ieee.org/db/oui/tutorials/EUI64.html

João Cândido Dovicchi obtained a M.A. degree in Arts in 1993 at Communication and Arts Institute of the University of São Paulo (USP, Brazil) and D.Sc. in Electrical Engineering in 1999 at Uberlândia Federal University (UFU, Brazil). His research interest is in Digital Signal Processing for Multimedia and Network Communication. He is professor and researcher at Uberlândia Federal University and directs the Advanced Center for Computing in Sound and Multimedia (http://www.nacs.demac.ufu.br)

João Bosco da Mota Alves obtained a degree in Electrical and Electronic Engineering in 1971 (UFPA, Brazil), a M.Sc. in Electrical Engineering in 1973 (UFSC, Brazil) and a Ph.D. in 1981 (COPPE/UFRJ, Brazil). After graduating he was employed as Professor at Federal University of Para, Federal University of Uberlandia, and Federal University of Santa Catarina, in Brazil. During this time his main interests were in Intelligent Automation and Robots. He is the Director of RExLab - Remote Experimentation Lab at Federal University of Santa Catarina (http://rexlab.inf.ufsc.br/).