Designing and Testing IPv6-enabled Networking Software

Mauro Tortonesi
Deep Space 6

mauro@deepspace6.net

The adoption of new communication protocols is quickly changing the Internet infrastucture. In this dynamic scenario, it is extremely important to upgrade existing services to the new protocols. The deployment of IPv6, in particular, creates new challenges for software designers and developers. The next generation of networking applications will have to support the IPv6 protocol and ensure backward compatibility with the IPv4 protocol. In addition, the applications will be flexible and highly configurable to work in the wide range of mixed IPv4 and IPv6 environments that will coexist during the transition to IPv6. This paper describes the problems that may arise in porting applications and services to IPv6, advocates the adoption of an AF-independent development style for software based on the BSD Socket API, and presents a set of tools to ease the process of developing and testing IPv6-enabled applications.

1. Introduction

The diffusion of broadband connectivity, the wide range of devices (such as smart phones and PDAs) together with multimedia applications (such as video on demand and voice over IP) calls for the adoption of new communication protocols with enhanced addressing and scaling capabilities, and compatibility with the Internet Protocol Suite. To address these problems, in the last years the Internet Engineering Task Force (IETF) and the whole research community have developed the IPv6 protocol and have made significant efforts to speed up the deployment of IPv6 networks. The experimental phase of IPv6 is coming to an end and the protocol is finally ready for mainstream adoption. The major networking equipment vendors are upgrading (the firmware of) their devices to support IPv6. The Internet infrastructure is evolving quickly towards IPv6 [DOERING]. ISPs have gathered important experience from the involvement in many IPv6-related research projects (especially in Asia and Europe) and are starting to offer IPv6 connectivity and IPv6-based services to their customers.

Unfortunately, all the existing TCP/IP networking applications must be modified to take advantage of the new communication protocol. Porting applications to IPv6 is a difficult task because of backward compatibility, portability, and the heterogeneity of the networking scenario. In fact, IPv6-enabled applications must preserve interoperability with all the IPv4-only services that do not support the new protocol yet. In addition, portable applications must often cope with all the peculiarities of systems that feature IPv6 support but are not fully compatible with the latest standards defined by IETF. Finally, since many different scenarios of mixed IPv4 and IPv6 connectivity will coexist during the transition, applications must be designed to work in a wide range of possible environments.

This paper focuses on the problem of porting applications and services to IPv6. The requirements of backward compatibility and portability, and the heterogeneity of the network scenario call for a careful development methodology, which takes into account security, code maintainability, support for other communication protocols and possible interactions with deployed transition tools and mechanisms. The paper discusses the issues that have to be addressed in the design of IPv6-enabled applications; it then introduces the Extended BSD Socket API, which is the standard tool for developing IPv6-enabled applications, and advocates the AF-independent development style as a "best practice" approach in the development of modern IPv6-enabled networking applications.

The methodologies and design criteria proposed in this paper have been applied in the development of many IPv6 applications and services. Experience has showed that the development and testing of IPv6-enabled applications can be made easier by using the libds6, nc6 and spak6 tools, presented later in this paper.

2. The IPv6 protocol

IPv6 is the new version of the Internet Protocol, devised by IETF to replace IPv4 and overcome its structural limits: address space exhaustion, explosion of routing tables and lack of support for mobility.

In fact, the imperfect IPv4 address allocation policies adopted before the introduction of CIDR have led to a enormous waste of addresses, of such an order of magnitude that the whole IPv4 address space will be soon completely exhausted. Besides, the IPv4 address space is a flat and not aggregatable one, and this constitutes both a very resource-hungry requirement for routers and a destabilizing factor that can lead to poor performance in routing decisions and convergence problems in the routing protocols. In addition, with regard to support for the mobility of portable devices, Mobile IP is not a scalable solution, since it requires the deployment of Foreign Agents in visited networks, it is very inefficient from the routing point of view, and its adoption is hindered by the widespread use of ingress filtering practices and Network Address Translators.

IPv6 is a long term solution for these problems: its enormous, aggregatable address space provides excellent addressing capabilities and prevents an uncontrolled growth of the routing tables, and the CIDR-like policy adopted for IPv6 address allocation reduces the waste of addresses to a more than acceptable level.

IPv6 has also robust autoconfiguration capabilities that minimize the need of administrative work in site renumbering and in address assignment to network interfaces, introducing Stateless Address Autoconfiguration as an alternative to stateful address autoconfiguration (DHCPv6).

Finally, Mobile IPv6 represents a scalable, low cost solution for mobile networking, since there is no need to deploy mobility agents in visited networks, as the roaming of mobile devices is automatically supported by any network that makes use of Stateless Address Autoconfiguration, and routing performance has been greatly improved.

Although IPv6 is superior to IPv4 in almost every aspect, since an incalculable number of hardware devices and software applications must be upgraded to make use of the new protocol, it is a common opinion that the transition from IPv4 to IPv6 will be long (perhaps more than a decade) and difficult.

3. Possible deployment scenarios

IPv6-enabled applications must dynamically adapt to the networking environment in which they run and support all the available communication protocols. This requires a careful design of applications and services, which can only be achieved if the developers have a deep knowledge of the scenarios in which the software will be deployed. During the transition to IPv6, many different environments with mixed IPv4 and IPv6 connectivity will coexist. There will be environments where:

- nodes have IPv6 connectivity but no IPv4 connectivity;
- nodes have IPv4 connectivity but no IPv6 connectivity;
- nodes have both IPv4 and IPv6 connectivity (dual stack nodes).

Notice that the first category includes also dual stack nodes with IPv4 connectivity disabled and the second category includes also dual stack nodes with IPv6 connectivity disabled.

There will be cases in which IPv4 connectivity is to be preferred to IPv6 connectivity or viceversa. This may happen if a network has two separated links to the outside Internet, one for IPv6 and the other for IPv4, which have different cost or reliability. In the first phases of the transition, it is likely that many hosts will not have native IPv6 connectivity, but will instead use one of the transition mechanism designed by IETF (such as Tunnel Broker, 6TO4 or NAT-PT) to get connectivity to the IPv6 Internet. This may create single points of failure in the IPv6 network and make IPv6 connectivity unstable, letting communication over IPv4 be the preferred choice for mission-critical applications.

During the transition phase, there may also be problems with DNS name resolution. Although the IETF discourages the practice of publishing a DNS AAAA record for a node before all of its services have been ported to IPv6, we will have cases that will break this rule, e.g., dual stack nodes with both an A and an AAAA record which will have at least some services only available via IPv4.

IPv6-enabled applications must cope with all these problems. This paper assumes that all the applications will be deployed on dual-stack hosts.

4. Design of IPv6-enabled applications

The next generation of networking applications should be able to communicate over both the IPv4 and IPv6 protocols. In fact, having two different applications (or versions of the same application) to handle networking services, one for IPv4 and the other for IPv6, may cause problems. On the server side there could be inconsistencies that may be very difficult to address, such as dual stack client applications that connect once to the IPv6-only application and the next time to the IPv4-only application in an unpredictable fashion. On the client side it could be annoying for the users, who must be aware that one application is IPv6-only and the other is IPv4-only, and it may force the sysadmins to deploy wrapper applications. In addition, the applications must be designed to work even if the target hosts have IPv4 or IPv6 connectivity (or even support) disabled.

Applications should also be built with flexibility in mind, because they should work in a wide range of different scenarios. In fact, there will be cases in which an application should work only with a given protocol version (IPv4 or IPv6) or with both of them. Such cases may happen if, e.g.,:

- a user is connecting to a service which is based on two different applications, e.g., the service is based on an IPv4 server app and an IPv6 Application Level Gateway or bouncer;
- a developer wants to test IPv6 compliance of a server application;
- a developer wants to test an IPv6 transition tool or mechanism;
- a user knows that the service he is trying to connect to is available only via a specific protocol version and wants to speed up the procedure to establish a connection to the server node.

IPv6-enabled applications should allow the expert user to choose if he wants to use IPv4, IPv6 or both.

Client applications should handle possible connectivity problems (even the ones which are due to bad DNS configuration) in a robust way. When connecting to a given hostname which resolves to more than one IP address, the application should try connecting to the first address returned by the resolver and should handle possible failures by trying to connect to another address [SHIN].

If the application performs DNS caching, it must cache both A (IPv4) and AAAA (IPv6) DNS records and discard the cached records as soon as their lifetime expire. Since many problems can arise from the interaction of the DNS caching pratice and the use of dynamic DNS, the expert user should be allowed to disable application-level DNS caching.

From the security point of view, the applications should handle potential problems related to the malicious use of IPv4-mapped IPv6 addresses on the wire [HAGINO1] [HAGINO2].

When porting an already existing application to IPv6 it may be desirable, e.g., for portability towards older systems which do not support IPv6 yet or to release an IPv6-enabled development version of the application while retaining production-quality IPv4 support code, to keep also the old IPv4-only source code and choose at build time if the application must use the old IPv4-only code or the new IPv6-enabled code.

5. The Extended BSD socket API

The IETF has developed an Extended BSD socket API [RFC3493] [RFC3542] in order to introduce support for the IPv6 protocol in the widely used BSD socket API, which was incompatible with IPv6 for three reasons: the sockaddr_in and in_addr data structures are inadequate to store IPv6 addresses; the inet_ntoa(3) and inet_aton(3) functions are inadequate for the conversion of IPv6 addresses from network to ASCII string format and viceversa; and the gethostbyname(3) and gethostbyaddr(3) functions cannot perform lookup (both forward and reverse) of IPv6 addresses in the Domain Name System.

To extend the core functions, e.g., socket(2), connect(2), bind(2), in order to handle IPv6 connectivity, the Extended BSD socket API defines the new protocol family PF_INET6 (with the related address family AF_INET6) and introduces the in6_addr and sockaddr_in6 data structures to store IPv6 addresses. To preserve backward compatibility, PF_INET6 sockets do not support only IPv6, but also IPv4. Connection to an IPv4 server application via a PF_INET6 socket is supported by means of IPv4-mapped addresses (IPv6 addresses with an embedded IPv4 address, identified by the prefix 0:0:0:0:0:FFFF::/96). A call to connect(2) with a PF_INET6 socket and the ::FFFF:194.243.234.21 IPv6 address as arguments will establish an IPv4 connection to the IPv4 host identified by the 194.243.234.21 address. In a similar way, an IPv6-enabled server application which binds to the IPv6 unspecified address (::) will also bind to the IPv4 unspecified address (0.0.0.0) and will thus accept incoming connections via both IPv4 and IPv6. Developers can change this default behaviour by setting the IPV6_V6ONLY socket option for PF_INET6 sockets. In this way, IPv4 compatibility is turned off and the PF_INET6 socket will support only IPv6.

The Extended BSD socket API also defines two new functions for conversion of IP address formats from network to ASCII string format and viceversa: inet_ntop(3) and inet_pton(3).

With regard to name resolution, the Extended BSD socket API defines two new functions: getaddrinfo(3) for DNS forward lookup and getnameinfo(3) for DNS reverse lookup. getaddrinfo translates a location, e.g., a hostname, and/or a service name and returns a set of socket addresses that can be used to connect or bind to the specified service. getnameinfo, instead, translates the location/service couple contained in a socket address structure to a node name and/or service name.

Developers can direct the operation of getaddrinfo and limit the set of addresses returned by that function to a specific socket type, address family and/or protocol by acting on the addrinfo structure which the function accepts as a configuration parameter.

6. Writing AF-Independent Applications

Although developers may be tempted to port their applications to IPv6 by simply changing all the occurrences of AF_INET and sockaddr_in to AF_INET6 and sockaddr_in6, in most cases this is not the approach that gives the best results. In fact, this hardcoding practice undermines the portability of the code, complicates the task of writing protocol-dependent services like FTP and prevents applications from working properly on dual stack systems where the IPv6 support is disabled [HAGINO3].

Instead, this paper recommends the adoption of an AF-independent development style, which can be achieved by making an appropriate use of getaddrinfo and getnameinfo. In fact, those functions have been designed to perform name-to-address and address-to-name resolution for all the communication protocols supported by the system (even those which are not based on DNS); thus passing a location and a service name to getaddrinfo returns all the addresses that can be used to connect or bind to that specified service, and passing a socket address to getnameinfo returns the correspondent location and service name.

In this way, applications will automatically take advantage of other protocol families, e.g., PF_UNIX, and communication protocols, e.g., AppleTalk or SCTP/IP, supported by the target host. In addition, applications will also be able to support new communication protocols as they will be implemented in future, without changes to the source code and without even recompiling it.

Although writing AF-independent code is usually rather easy, problems may arise from the fact that AF-independent development style requires setting the IPV6_V6ONLY socket option of PF_INET6 sockets, and not all the systems support that option in the same way. In fact, although IETF recommends ISVs to turn the IPV6_V6ONLY socket option off by default, some systems, e.g., NetBSD, OpenBSD, FreeBSD 5.0 and later, adopt the opposite behaviour and other systems allow system administrators to choose the default behaviour at run time, e.g., starting from kernel version 2.4.21, Linux exports the system wide sysctl configuration option /proc/sys/net/ipv6/bindv6only. Moreover, many systems do not support the IPV6_V6ONLY socket option yet. This inconsistence makes the task of writing portable AF-independent applications more difficult, especially for server applications.

7. Higher level APIs for writing networking applications

The BSD socket API was the first API ever released for writing TCP/IP networking applications. Introduced by BSD 4.2 and later adopted by nearly all Unix operating systems, it has then been standardized by POSIX and is now available on nearly all platforms. Although it is a rather low level API, it still very widely used because of its incomparable portability and flexibility, and because many developers are familiar with it.

However, there are many other software tools that can be used to write networking applications. The API exported by networking libraries and middleware can generally be divided in two categories: those based on the socket concept and those based on Remote Procedure Calls or Remote Method Invocation.

The APIs provided by the first category of these tools, e.g., Perl Socket6 and IO::Socket::INET6 modules, Python socket module and Java sockets, are usually a higher level version of the BSD socket API itself. With regard to the problem of porting to IPv6 the networking software written using these tools, the same considerations presented in the first part of the paper apply.

Instead, porting distributed applications based on Remote Procedure Call, e.g., ONC RPC, XML-RPC and SOAP, and Remote Method Invocation, e.g., CORBA and Java RMI, middleware usually doesn’t require any modification to the program sources, as it is the RPC or RMI infrastructure itself (almost always implemented in C or C++ using the BSD socket API) that must be modified in order to support the IPv6 protocol.

8. Testing IPv6-enabled software

Since the IPv6 support of some systems is still at an experimental stage, extensive testing of IPv6-enabled networking code is of great importance. In addition to common debugging and testing practices, this paper recommends using specific tools to verify the conformance to the IETF standards of the Extended BSD socket API implementation in the target systems.

For this purpose, we have developed libds6, a package that contains a set of programs and library functions to check if the IPv6 support on the development platform is correctly implemented and to inspect networking code data structures. The libds6 package contains the testgetaddrinfo, testgetnameinfo, getaddrinfo and getnameinfo tools and the libds6 library. The testgetaddrinfo and testgetnameinfo programs check if the implementation of the getaddrinfo and getnameinfo functions on the local system is compliant to the latest IETF standards. The getaddrinfo and getnameinfo programs are instead tools for performing querys to the DNS system by using the getaddrinfo and getnameinfo functions. Finally, the libds6 library provides many routines that can be used to inspect the contents of socket addresses, addrinfo structures and other networking code internal data structures. The information provided by the programs and library functions contained in the libds6 package can be very helpful for developers to check if the implementation of the Extended BSD socket API provided by their target system is buggy and if it allows the adoption of the AF-independent development style.

Another application designed to ease the process of testing IPv6-enabled applications and services is nc6, a command-line utility which reads and writes data across network connections. The nc6 application can be used to create a TCP connection or send data via UDP to a given port on a given target host, using either IPv4 or IPv6 as appropriate. The data received by the application on its standard input is then sent to the remote host, and anything that comes back across the connection is sent to the application's standard output. nc6 can also work in server mode, by listening for inbound connections on arbitrary ports and then doing the same reading and writing when a client connects. Since nc6 provides many advanced features, it can be of great help to test both client and server applications implementing text-based protocols, such as HTTP, FTP, SMTP, and all TELNET-based protocols.

spak6, instead, is a powerful, low level packet forger that allows the creation and the injection of hand-crafted IPv4 and IPv6 packets in the network, and has been designed for the purpose of testing both distributed applications and networking protocol stack implementations.

These tools have proved to be of great help in the development of IPv6-enabled applications and can considerably ease the process of testing IPv6-enabled networking software. The source code of libds6, nc6 and spak6 is available for download at the Deep Space 6 source code page (../sections/sources.html).

9. Conclusions

The transition to IPv6 will require the porting of existing applications and services to the new protocol, a very hard task given the significant constraints of backward compatibility and portability, and the heterogeneity of the networking environments in which applications will be deployed.

This motivates the research efforts in both industry and academia. Research projects like 6NET and EURO6IX (both funded by the EC) have produced so far many interesting results, and have been an ideal testbed for deploying experimental IPv6-enabled (and even IPv6-specific) applications and services, and to test in practice existing transition tools and mechanisms. However, a lot of work has to be done, as there is the need of further studies on the development methodologies for writing extensible, change-resistant IPv6-enable code and of new, advanced tools to automate and speed up the process of porting applications and services to IPv6.

In this context, the article advocates the AF-independent methodology for the development of IPv6-enabled applications and services. The paper also presents libds6, nc6 and spak6 as precious helper tools for the testing and debugging of IPv6-enabled applications and services.

Now that a significant portion of the Internet has IPv6 connectivity, we should expect an increase in the number of applications and services which will be available via IPv6. It seems that, after many long years of waiting, IPv6 is finally coming.

References

Gert Doering, *An overview of the global IPv6 routing table (http://www.space.net/~gert/RIPE/R49-v6-table/)* .

M. Shin, Y. Hong, J. Hagino, P. Savola, and E. Castro, *Application Aspects of IPv6 Transition (http://www.ietf.org/internet-drafts/draft-ietf-v6ops-application-transition-03.txt)* .

Craig Metz and Jun-ichiro Hagino, *IPv4 mapped address on the wire considered harmful (ftp://ftp.itojun.org/pub/paper/draft-itojun-v6ops-v4mapped-harmful-02.txt)* .

Jun-ichiro Hagino, *Possible abuse against IPv6 transition technologies (http://playground.iijlab.net/i-d/draft-itojun-ipv6-transition-abuse-01.txt)* .

Craig Metz and Jun-ichiro Hagino, *IPv4 mapped address API considered harmful (ftp://ftp.itojun.org/pub/paper/draft-cmetz-v6ops-v4mapped-api-harmful-01.txt)* .

Robert Gilligan, Susan Thomson, Jim Bound, Jack McCann, and W. Richard Stevens, *Basic Socket Interface Extensions for IPv6 (http://www.ietf.org/rfc/rfc3493.txt)* .

W. Richard Stevens, Matt Thomas, Erik Nordmark, and Tatuya Jinmei, *Advanced Sockets Application Program Interface (API) for IPv6 (http://www.ietf.org/rfc/rfc3542.txt)* .