Programming guidelines on transition to IPv6

Tomás P. de Miguel and Eva M. Castro
tmiguel@dit.upm.es
eva@gsyc.escet.urjc.es

Department of Telematic Systems Engineering (DIT)
Technical University of Madrid (UPM)

Systems and Communications Group (GSyC)
Experimental Sciences and Technology Department (ESCET)
Rey Juan Carlos University (URJC)

January 2003, v13

Acknowledgements

This research has been supported by LONG (Laboratories Over Next Generation networks) project, IST-1999-20393.

We would like to thanks Latif Ladid and Jim Bound for their support and encouragement. Thanks to the people who made suggestions and provided feedback to this document specially Jordi Palet, which improve the accuracy and presentation of the text.

PROGRAMMING GUIDELINES ON TRANSITION TO IPV6 1

ACKNOWLEDGEMENTS 2

FIGURES 4

TABLES 4

1. INTRODUCTION 5

2. TRANSITION TO IPV6 WITHOUT CHANGING APPLICATIONS 6

3. TRANSITION SCENARIOS 8

4. PORTING APPLICATIONS 11

4.1 ANALYZING EXISTING PROGRAMS 11
4.1.1 Application transport module *12*
4.1.2 Other application modules with IP address dependencies *13*
4.2 IPV4/IPV6 INTEROPERABILITY 17
4.2.1 IPv6/IPv4 clients connecting to an IPv4 server at IPv6-only node *17*
4.2.2 IPv6/IPv4 clients connecting to an IPv6 server at IPv6-only node *18*
4.2.3 IPv6/IPv4 clients connecting to an IPv4 server at dual stack node *19*
4.2.4 IPv6/IPv4 clients connecting to an IPv6 server at dual stack node *20*
4.2.5 IPv4/IPv6 clients connecting to an IPv4-only server and IPv6-only server at dual stack node. *21*
4.2.6 Client/server and network type interoperability *21*
4.3 PORTING SOURCE CODE 22
4.3.1 Socket Address Structures *23*
4.3.2 Socket functions *24*
4.3.3 Required modifications when porting to IPv6 .... *25*
4.3.4 Address conversion functions *28*
4.3.5 Resolving names *30*
4.3.6 Multicasting *32*

5. GUIDELINES ON DEVELOPING NEW APPLICATIONS 35

6. CONCLUSIONS 37

7. BIBLIOGRAPHY 40

8. APPENDIX: EXAMPLES OF REAL PORTING PROCESS 41

8.1 ORIGINAL DAYTIME VERSION 41
8.1.1 IPv4 Daytime server (TCP/UDP) *41*
8.1.2 IPv4 Daytime client (TCP/UDP) *46*
8.2 THE UNICAST DAYTIME PORTED TO IPV6 50
8.2.1 Daytime server (TCP/UDP) *50*
8.2.2 Example: Daytime client (TCP/UDP). *55*
8.3 THE MULTICAST DAYTIME 59

Figures

FIGURE 1: APPLICATION INTERFACE TRANSLATORS. 6
FIGURE 2: TRANSITION SCENARIOS. 9
FIGURE 3: APPLICATIONS PROTOCOL INDEPENDENT OF IPV4 OR IPV6. 13
FIGURE 4: TRANSFER UNIT (TU) SIZE PROBLEM IF NOT PMTU-D IMPLEMENTED. 16
FIGURE 5: IPV4/IPV6 CLIENTS CONNECTING TO AN IPV4 SERVER AT IPV4-ONLY NODE. 18
FIGURE 6: IPV4/IPV6 CLIENTS CONNECTING TO AN IPV6 SERVER AT IPV6-ONLY NODE. 19
FIGURE 7: IPV4/IPV6 CLIENTS CONNECTING TO AN IPV4 SERVER AT DUAL STACK NODE. 20
FIGURE 8: IPV4/IPV6 CLIENTS CONNECTING TO AN IPV6 SERVER AT DUAL STACK NODE. 20
FIGURE 9: IPV4/IPV6 CLIENTS CONNECTING TO AN IPV4-ONLY SERVER AND IPV6-ONLY SERVER AT DUAL STACK NODE. 21

Tables

TABLE 1: SPECIAL ADDRESSES. 14
TABLE 2: CLIENT SERVER AND NETWORK TYPE COMBINATIONS. 21
TABLE 3: MACROS FOR TESTING TYPE OF ADDRESSES. 32
TABLE 4: MULTICAST SOCKET OPTIONS. 32
TABLE 5: IPV4 OR IPV6 ACTIVATION. 36

1. Introduction

The transition between today's IPv4 Internet and the future IPv6-based one will be a long process during which both protocol versions will coexist. Moving from IPv4 to IPv6 is not straightforward and guidelines to simplify transition between the two versions have to be standardised. Network transition has been discussed in detail; however applications should be reviewed to complete the porting process.

Existing applications are written assuming IPv4. Only very recently IPv6 has been taken into account. Unless most of basic distributed applications are available now; there is too much work to do yet. The aim of this paper is to provide general recommendations to be taken into account during the porting process of applications and services to IPv6. This will allow developers to move smoothly their applications into the new environment.

The document is divided in three parts. The first analyzes in which conditions is possible the transition to IPv6 without changing applications. This chapter includes recommendations on how to proceed when source code is not available and explains which mechanisms can be used.

The second is the main document part. It starts describing IPv6 transition scenarios from the application point of view. The paper is focused on analyzing existing applications looking for characteristics, which usually should be reviewed during transition to IPv6. It includes a brief description of basic socket interface extensions for IPv6, fully described in RFC2553 [BISE].

The paper concludes providing general recommendations for new IPv6 applications. In the future all IPv4 networks will be IPv6; however during a long period mixed scenarios with both IPv4 and IPv6 will be the real environments. Therefore, new applications should be designed to work only in a pure IPv6 environment, but a design to allow mixed IPv4 and IPv6 environment is better now.

The document includes some examples of code porting process, used to illustrate the required changes in the client and server components. Porting guidelines are valid for any programming language, however for simplicity, application examples are provided only in C language. All these examples have been tested in a SuSE Linux 7.3 distribution, kernel version 2.4.10. The LONG Project has revised detailed rules and socket interface description for other popular programming languages [LD31].

2. Transition to IPv6 without changing applications

Many methodologies have been studied to support transition to IPv6 depending on initial network architectures. To make IPv6 network available to the user it's necessary to provide a great number of IPv6 applications. Although, existing applications are written assuming IPv4, it isn't desirable to wait until all necessary applications will be ready to start network transition.

The simple approach during initial transition stages is to use IPv4 applications over the new IPv6 network environment. Unfortunately, this is not a simple task. A pure IPv6 node provides only the IPv6 socket interface but old applications only work over IPv4 socket interface. Both interfaces are similar, but there are some incompatible differences related with:

- Name (DNS) and IP addresses management,
- Communication constants and data structures or
- Functions names and parameters

Therefore, the IPv4 stack should be available to continue using IPv4 applications. The network is IPv6, so to operate with the network the stack which should be used is the IPv6 one. Them dual stack is required.

But dual stack is not enough. If both IPv4 and IPv6 network interfaces are available but only there are IPv4 applications, IPv6 will not be used. A real IPv6 scenery implies removing IPv4 network and operating only through IPv6 one. The solution is to provide a translator from the IPv6 network interface into the IPv4 programming interface required by applications; see Figure 1.

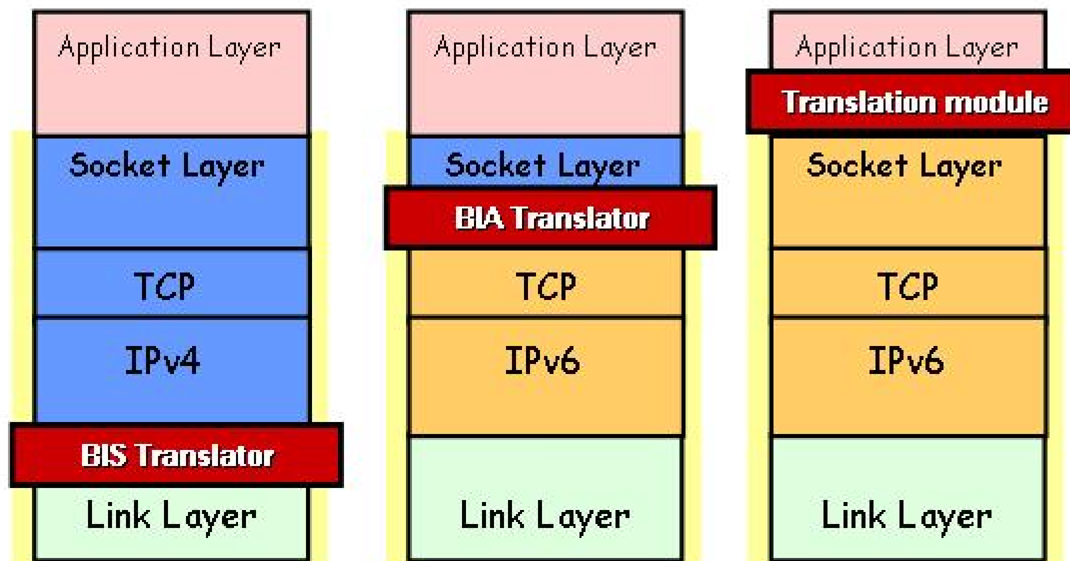

Figure 1: Application interface translators.

A translator intercepts data flowing between application and the link layer modules and translates all IPv4 related issues into IPv6 and vice versa. When IPv4 application communicates with an IPv6 node, the translator changes addresses into IPv6, changes the call into the new environment and outputs an IPv6 packet. The reception is similar. All

input packets are examined, translated and emitted to the upper layer in the IPv4 format. Thus it seems as if it was dual stack host with both IPv4 and IPv6 or dual applications. Translation can be made at three different levels: link, socket or application.

Link translation [BIS] takes place between IP level and network driver modules (the link layer). At this level the translator intercepts all input and output IP packets. If correspondent node is IPv4 the packet is emitted without changes. If correspondent node is an IPv6 one, it is revised and translated into an IPv4 packet. If it is an output packet, the translator obtains the IPv6 address of the correspondent node and outputs an IPv6 packet. When the node receives an IPv6 packet, it obtains the correspondent node IPv4 address (or builds it) and translates the original IPv6 packet into the IPv4 one.

Socket interface translation [BIA] takes place between socket API module and the TCP/IP module. Now translation is at interface programming level, so translation is simplified without translating IP packet headers. To communicate with an IPv6 node, the translator intercepts all IPv4 sockets calls and translates them into IPv6 sockets functions. The returns from IPv6 calls (packets reception) are translated into the correspondent IPv4 returns. The mechanism is complemented with translations of addresses and names, similar to the previous one.

Finally, sometimes applications use user level communications library, for example when a prototype or an interpreted programming language is used. In such a case, it is possible to introduce the translator at application level. The communications API module remains unchanged but it is rewritten in order to transform part of the IPv4 connections into IPv6 ones. Translation of names and addresses are also required to provide only IPv4 information to the application when it connects with an IPv6 node.

Unfortunately, there are many applications without a clear modularization or have addresses dependencies, in such cases application code should be revised to integrate the application in the new IPv6 environment.

3. Transition scenarios

The most important requirement in IPv6 transition is that existing services should work in the new environment but also continue to work with IPv4 nodes. The simplest approach to introduce IPv6 without changing applications is to use dual-stack: supporting IPv4 and IPv6 simultaneously, maintaining old IPv4 applications and adding new ones to communicate with IPv6 nodes. This largely increases the complexity of network administration and maintains the address space scarce resource.

There is a pressure to start up new networks in the new IPv6 environment without IPv4 support. However, there is no a single global IPv6 network on the same scale as the actual IPv4 and it will take some time to get it. Therefore, new applications should be designed to work in all environments: single IPv4, single IPv6 or mixed environment connecting IPv6 with IPv4 nodes. That is, dual protocol stack is necessary. However, it does not mean supporting simultaneously IPv4 and IPv6 routing. It is possible to use IPv6 applications on IPv4 network and IPv4 applications on IPv6 network.

Therefore, there are many possible scenarios but only some of them will be used in practice [APPT]. To select the best one it is necessary to know the initial environment. Two possible environments can be considered: the transition of working IPv4 network and the setup of new network or service.

For already working networks the better solution is to maintain the IPv4 stack and introduce IPv6 stack in parallel with the old one; that is to use the dual stack environment [DSTM]. It is the principal building block during transitioning from IPv4 to IPv6. Dual stack mechanisms do not, by themselves, solve the IPv4 and IPv6 interworking problems; other important building block, addresses translation, is required many times. Translation refers to the direct translation of protocols, including headers and sometimes protocol payload. Protocol translation often results in features loss. For instance, translation of IPv6 header into an IPv4 header will lead to the loss of the IPv6 flow label. Translation can be complemented with tunnelling; which is used to bridge compatible networks across incompatible ones.

After transition period, IPv4 stack and IPv4 applications will be removed and result a pure IPv6 environment.

The organizations transition from IPv4 to IPv6 follows more or less the same steps. Each step defines one transition scenario; see Figure 2. The systems manager should provide applications to work in all of them. We can consider various scenarios in application porting process using dual-stack:

Start point: Applications have been designed on IPv4 and the network is working with IPv4 only.

Step I: The network remains unchanged. Some IPv6 applications are available but IPv6 communication is only possible through IPv6 over IPv4 tunnels.

Step II: IPv6 is available at network level unless applications remain unchanged and only support IPv4. Only IPv6 applications can use the IPv6 network.

Step III: Applications are ready to work simultaneously on IPv4 and IPv6.

Step IV: IPv4 network has been removed. Communication with IPv4 nodes is only possible with the help of an IPv4 over IPv6 tunnel.

End point: Finally all IPv4 networks and applications use IPv6.

The application porting process takes place between steps II and III. This paper is devoted to describe how to make such application adaptations. There are two probable scenarios during the transition. The first is based on maintaining two application versions and the second with only one dual version.

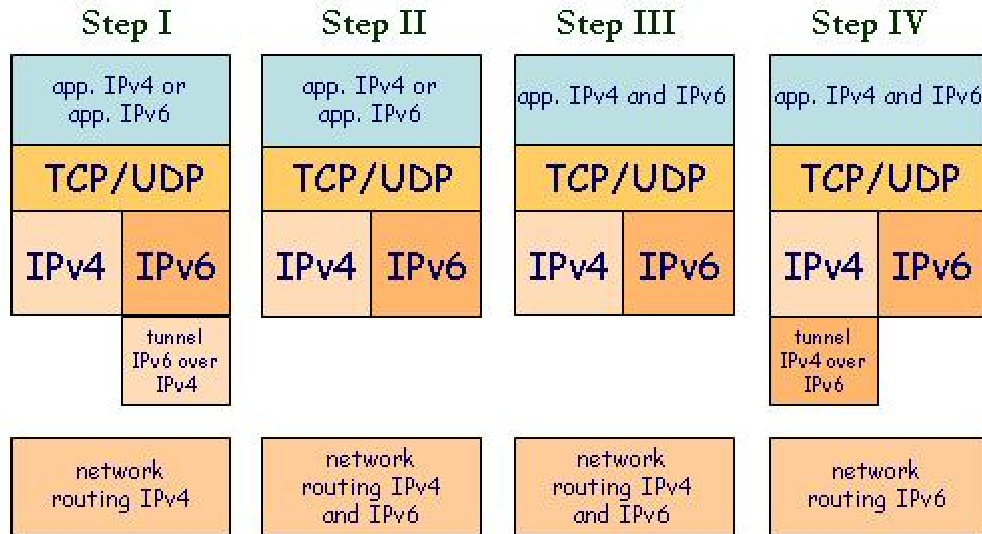

Figure 2: Transition scenarios.

We can forget old IPv4 applications and maintain them only to communicate with IPv4 nodes during transition period. The transition methodology consists in developing a complete set of applications designed to work only over IPv6 transport layer. The main advantage is to be ready with new applications after transition period. To finish IPv6 transition it is necessary only to removed IPv4 applications and dual stack network. However, the selection of suitable applications during transition period it's a user responsibility. When tests, configuration or management of applications are considered, the application selection is not a problem because their users normally have technical knowledge. However, with most of end-user applications, the user hasn't enough knowledge or doesn't want to know, which is the proper application version that should be used in each connection. Moreover, during transition period it will be necessary to provide data servers working both IPv4 and IPv6 to accept connections not only from new but also from old nodes.

The second transition scenario looks for changing existing applications, not developing new ones. The porting process takes the original IPv4 application, review source code and produces a new version adapted to work over both IPv4 and IPv6 environments. The porting process is a little bit more complex than IPv6 only support, but the result is more flexible to be used during transition period. Transition period will be probably a long period during which there will be islands, not only IPv6 but also IPv4, interested for users

in the new environments. This is because during a long period most of data servers will be maintained in the IPv4 only environment.

However, for new network or services an only IPv6 stack environment is better. It is not sense to define an IPv4 network so dual stack is not allowed. However, during transition period native IPv6 networks will need to maintain connectivity with applications, which can only be reached through IPv4. In such cases, one standard network transition mechanism should be provided [NAT-PT] [SIIT] [6to4]. Depending on the application, sometimes it is possible to use special dual-stack edge servers at the IPv4 and IPv6 border as application level proxies, removing the need for network-level transition.

4. Porting applications

Unless there are many proposed solutions to accelerate transition from IPv4 to IPv6, ultimately, applications should be modified to simplify protocol stacks configurations and support all IPv4 and IPv6 communication types.

Most existing applications are written assuming IPv4. In general, most of them can be converted without too much effort. Many changes could be done automatically and there are some scripts to do it. However, there are applications, which make special use of IPv4 or include advanced features, such as multicasting, raw sockets or other IP options. In such cases, an exhaustive code revision should be done.

Besides, there are applications that are not only affected by modification of function calls but also by the logic behind their usage. Such code is the hardest to deal with and cannot be automatically ported. This code can be misleading if developers do not consider the logic of the program during the porting process.

Dual stack allows applications to use both types of addresses, IPv4 and IPv6. Developers will only need to port their IPv4 applications to the new IPv6 API, considering that applications could communicate using both protocol versions, depending on the destination node.

In the general case, porting an existing application to IPv6 requires to examine the following issues in the application source code:

- Network information storage: data structures.
- Resolution and conversion address functions.
- Communication API functions and pre-defined constants.

Many applications use IP addresses to store references to remote nodes. This is not recommended because IPv6 lets addresses change over time (this technique is named renumbering). Applications should use stable identifiers for other nodes, for instance, host names. If applications store IP addresses they should redo the mapping from host names to IP addresses in order to solve inconsistencies.

Client applications should be prepared to connect to multi-homed severs, nodes that have more than one IP address. Hence, when a communication channel to a multi-homed server fails, client applications should try another IP address in the multi-homed server list of IP addresses until they find one that is working.

Applications using URLs should change definitions in RFC2732 which specifies square brackets to delimit IPv6 addresses: http://[IPv6Address]/index.html

4.1 Analyzing existing programs

Once the IPv6 network is deployed, applications must be designed to work over it. Existing applications cannot use IPv6 without previous modifications. Hence, from the applications point of view, the IPv6 deployment requires changes in the existing applications and new communication design concepts to be included for the new ones.

When porting IPv4 applications to IPv6, there are different degrees to carry out this task. If applications only use basic communications facilities, developers only have to identify the

application communication module and change some functions in order to use the new IPv6 API. However, if IPv6 advanced facilities should be considered, such as quality of service or mobility, a redesign of some parts of the application must be done. In this last case, the porting process requires a complete knowledge of the application architecture and this process is similar to the design of a new application over IPv6.

Providing basic IPv6 communication facilities to an existing application is an easy task if the application architecture is well designed, with isolated functional modules, so that the transport module could be easily identified. In the other hand, if the application has not isolated modules, adaptation to the new API will require many changes in different parts of source code, making difficult the reusability and the maintenance.

The application transport module is charged of implementing the communication paradigm selected by the application and must be adapted to the new IPv6 communication API. Usually, the communication API makes visible to the application the version of protocol it is using, since address data structures are completely different. Besides, other facilities such as conversion functions between hostnames and IP addresses are different in the new IPv6 environment too.

Besides, there could be other modules or application parts different from transport module, which can include dependencies on the IP addresses, and they must be reviewed:

- IP address parsers.
- Use of special addresses.
- Local IP address selection.
- Application Data Unit (ADU) fragmentation.
- Register systems based on IP addresses.

In the following subsections, some modifications to be made in the transport module and in other modules, which include dependencies on the IP addresses, are studied.

4.1.1 Application transport module

When porting to IPv6, the first module to be reviewed is the one related to the management of the remote communication channels, named transport module. This module will surely contain dependencies on the IP version used by the application.

The application transport module establishes communication channels required by a specific application when transmitting data to remote sites. This module makes applications network independent, providing a generic communication module to allow applications exchange information between remote participants.

Changes in the application transport module are required since the kernel communication API has been changed to support IPv6. The module must be changed to use the new address structures and functions to create IPv6 communications.

During the gradual transition phases from IPv4 to IPv6, the same application will work over IPv4 and IPv6 networks. Hence, portability is one of the main features of applications, which should work in both environments. One of the best ways to make applications independent of the protocol used (IPv4 or IPv6) is to design a generic communication library, which hides protocol dependencies and lets the application transport module be simpler. The use of the library allows reusability and easy maintenance of the

communication channel abstraction. Applications can forget about communication problems, since the library charges on behalf of them, see Figure 3.

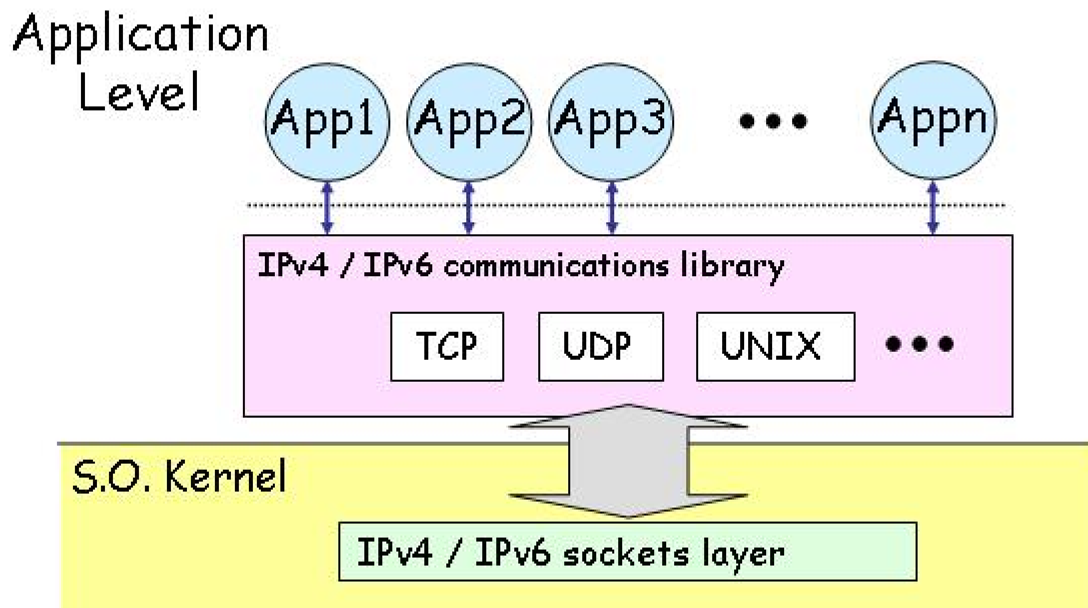

Figure 3: Applications protocol independent of IPv4 or IPv6.

A generic communication library allows applications to be independent of the lower level protocol, and provides a common communication interface to every application. Applications should be able to access to different communication protocols from the generic library API functions: TCP, UDP, IP, multicast, etc., independently of the IP version.

4.1.2 Other application modules with IP address dependencies

The application transport module is directly related to the communication establishment and the information transmission. However, there could be other application modules, which have IP address dependencies, so they are IP version dependant. In the next subsections these other dependencies are analysed.

4.1.2.1 IP address parsers

Many applications require an IP address as an argument to establish a new connection to this address, for example using the client/server model the client needs to know the IP address or the hostname where the server is running.

The use of Fully Qualified Domain Name (FQDN) instead of IP address is preferable since nodes can change their addresses and this process should be transparent to applications. Applications can store and use FQDN, delegating the resolution of the IP addresses to the name resolution system, which will return the associated IP address at the moment of the query.

From applications point of view, the name resolution is a system-independent process. Applications call functions in a system library, known as the resolver, which is linked into

the application when the application is built. Developers should change the use of the IPv4 resolution functions by the new IPv6 ones, or by the protocol-independent ones if they exist.

Typically applications do not require knowing the version of the IP version they are using, hence applications only should try to establish communications using each address returned by resolver until it works. However, applications could have a different behaviour when using IPv4, IPv6, IPv4-compatible-IPv6 or IPv4-mapped, etc. addresses.

There could be scenarios where the use of IP addresses is required and applications utilise a parser to analyse addresses. In these cases, IP address parsers must be modified in order to include the new IPv6 string address format.

IPv4 addresses are represented using dotted quad format, each decimal integer represents a one octet of the 4 octet address, value between 0 and 255; for instance 138.4.4.161. The written length of the IPv4 address string varies between 7 and 15 bytes. IPv6 addresses are represented using hexadecimal notation which can be abbreviated, requiring between 3 and 39 bytes; for instance 2001:720:1500:1::a100. So, IPv4 uses dot (".") to separate octets and IPv6 uses colon (":") to separate each pair of octets. Application address parser code should be reviewed to be in conformance with the IPv6 address representation.

There could be an ambiguity with the colon character. Colon character is used in the IPv6 addresses as a separator between each pair of address octets and it is used in the IPv4 networks as a separator between the address and the service port number. Applications can use the same format as the literal IPv6 addresses in URLs, enclosing the IPv6 address within square brackets, to solve the ambiguity. The RFC 2373 describes literal IPv6 addresses in URLs, for instance: http://[2001:720:1500:1::a100]:80/index.html

4.1.2.2 Use of special addresses

There are some special addresses, which are found hard coded within the application source, instead of using symbolic names. For instance, the addresses 127.0.0.0/8 is referred to the localhost interface. When moving an application from an only IPv4 host to an only IPv6 host, hard coded addresses will fail when establishing connections. So the use of names instead of IP addresses is recommended, names could be reconfigured without changing application source code.

Developers should review application source code to change the IPv4 addresses in Table 1 to consider portability to IPv6 only hosts.

Table 1: Special addresses.

Symbolic Name (IPv4 / IPv6)	IPv4 Address	IPv6 Address
`INADDR_ANY / IN6ADDR_ANY_INIT`	0.0.0.0	::
`INADDR_LOOPBACK / IN6ADDR_LOOPBACK_INIT`	127.0.0.1	::1
`INADDR_BROADCAST`	255.255.255.255	Not exist

4.1.2.3 Local IP address selection

IPv6 allows many IP addresses by each network interface with different reachability scope (link-local, site-local or global). Hence, there should be mechanisms to select which source

and destination addresses applications should use in order to know the behaviour of the systems.

Normally, the name resolution functions return a list of valid addresses for a specific FQDN. Applications should iterate this list to select the address to be used in the communication channel configuration (for instance, to select the source address, for bind() function, and to select the destination address, for `sendto()` or `connect()` functions). Source address selection is a critical operation that gives information to the receiver about the address to send the reply. If the selection is not appropriated the backward path could be different from forward path, even if the addresses are administratively scoped, the reply may be lost and communication between applications will fail.

When choosing source address, some applications use unspecific address to let the OS kernel make this selection, named default address selection. When choosing destination address, some criteria could be used to prefer one address based on the pair source/destination values. The default address selection algorithm returns a preferred address from a set of candidates, based on a policy to make the best choice. Administrators can configure this mechanism to override the default behaviour. There are still some works in progress around this topic (draft-ietf-ipv6-default-addr-select.txt).

4.1.2.4 ADU fragmentation

Application Data Unit (ADUs) is the block of data sent or received in a single communication operation at application level. Applications alternate between computation and communication phases, and during the communication phase they send or receive one or more data structures, each forming an ADU. However, since the amount of data that can be handled at any given time, in a single unit of operation is limited by the available resources on the network interface node, the ADU must be fragmented in transfer units (TUs).

The problem is to select the more adequate TU size. Long packets are transmitted more efficiently, as the application overhead of the end systems is reduced. On the other hand, longer packets tend to increase transit delays because of the intermediate relaying process, which is not good in real time applications. The size of the TUs is directly related to the maximum size of the IP packet used over a network (PMTU, Path Maximum Transmission Unit) and the IP fragmentation process. Then, longer packets are likely to be fragmented to adapt the packet size to the link layer.

Since IPv6 fragmentation is an end-to-end algorithm, [RFC 1981] recommends that all IPv6 nodes should implement PMTU Discovery (PMTU-D) to optimise the throughput of fragmented ADUs. PMTU-D is a mechanism to use the IPv6 packet size longer than fits the IPv6 minimum MTU through all the networks traversed, increasing the efficiency of transmission and guaranteeing that the IPv6 packets can travel through all networks to reach the destination. If a packet is too large for a router to forward on to a particular link, the router must send an ICMP "Destination Unreachable -- Fragmentation Needed" message to the source address an the source host adjusts the packet size based on the ICMP message.

However, the PMTU-D is implemented in the packetization protocol, but it is only a recommendation not a mandatory network module, and routers do not always rely on ICMP packets, see RFC2923. So, applications can send their TUs using a TU size longer than the MTU of an intermediate link on the current path, between the source and the

destination nodes and it is likely none of the packets will reach the destination node. This problem is known as the black hole. For example, in Figure 4, packets are not fragmented at the source because they can be sent over this first link layer. But, when the packets reach the intermediate link R_{inter}, which is using an outgoing MTU smaller than the packet size, packets will be discarded and will not reach destination node. In Figure 4 TU size is longer than the MTU3 of the intermediate link, at this point packet will be discarded.

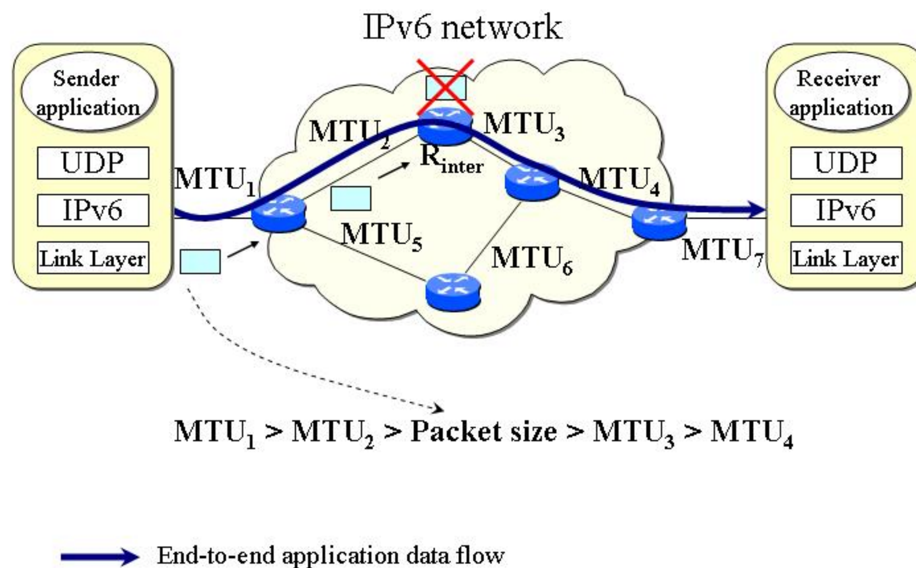

Figure 4: Transfer Unit (TU) size problem if not PMTU-D implemented.

TCP is a packetization protocol and some implementations solve this problem implementing black hole detection and sending smaller packets until the transmission is done.

However, there are applications, which realize their own packetization process, maybe sending UDP packets. If PMTU discovery is not properly working, data will not reach destination. In this situation, applications must implement their own mechanisms to detect black hole problem and send smaller packets, or use the minimum supported MTU for IPv6 1280 octet packets.

4.1.2.5 Register systems based on IP addresses

Analysing network applications, dependencies in the source code with the IP address are frequently found. For instance, one of the most popular ways to register remote nodes in a collaborative system is based on using the IP addresses as keys for searching in a registry system. Group communication is often related to a group membership concept based on a participant registry system.

The registry system provides an identification method to allow connections from different remote participants to a session. The problem is that an IP address cannot be used to

identify a peer node since IP addresses can change over time, for instance after a renumbering process. Renumbering should be an infrequent event, but sometimes it will happen and it should be a transparent process for applications.

The best solution to this problem is the use of identifiers independent of the network layer, for instance random numbers. However, some applications require that the identifiers were dependent on network address. One of the flexible solutions is the use of FQDN as stable identifiers for participant nodes in the registry system. The FQDN remains invariable over the time although it's associated IP address can change. The resolver code will provide the IP address if it was required at any time.

Sometimes applications require IP addresses instead of FQDN. When porting to IPv6, this dependency will provoke some changes in the source code to support the new data structures. In these cases, applications can get addresses from FQDN and store them in configuration files for an unlimited time. This information should be refreshed periodically to actualise IP address modifications and to avoid no longer valid information. The refreshing process will guarantee the use of the last assigned IP address to a node.

4.2 IPv4/IPv6 Interoperability

The mechanism to select the appropriate IP address is decided by the name service. When a network node wants to reach another, it asks the name service for its IP address. If the answer is an IPv4 address, it is assumed that there is a path through Internet to link with the remote node and that this remote node is capable of receiving IPv4 connections from the source node. The same applies for a node that has only IPv6 address. It is assumed that it understands IPv6 packets. If both source and destination nodes have dual stack, the communication will use the type of address returned by the name service.

During first step of transition phase there is no IPv6 network. This document is not devoted to study how to solve other communication aspects, which are not visible to the application layer. If IPv6 addresses should be used during connection but IPv6 routers are not part of the network infrastructure, a basic IPv4 framework should be used. This is achieved by building IPv6 tunnels. They encapsulate IPv6 packets inside IPv4 header and send them through the IPv4 network.

However, dual-stack should be used during most of the transition periods, because not all IPv6-only implementations allow the interaction with any kind of network node. Following we will study the interoperability between IPv4-only nodes, IPv6-only nodes and dual stack nodes using the client/server model.

4.2.1 IPv6/IPv4 clients connecting to an IPv4 server at IPv6-only node

There is an IPv4 server application running on IPv4-only node and we will analyze all connection possibilities from clients, see Figure 5.

If an IPv4 client running on IPv4-only node connects to the IPv4 server, it will work as the usual IPv4 client/server connection, since both of them are IPv4-only nodes.

When using an IPv4 client running at dual stack node, the client will request the server IP address to the resolver and it will return the server IPv4 address. The client running at dual stack will work as if it was running at IPv4-only node and will exchange IPv4 packets.

When using an IPv6 client running at IPv6-only node, since this node can only exchange IPv6 packets, it cannot connect to the server, which can only exchange IPv4 packets. They are using incompatible protocols.

If an IPv6 client is running at dual stack node, the client will request to the resolver the server IP address. Since the server is running at an IPv4-only node, the resolver will return the IPv4-mapped IPv6 address. The IPv6 client will use this address to connect to the IPv4 server and the dual stack node will send IPv4 packets when the IPv4-mapped IPv6 address is used. Then, IPv6 client will work as it was connected to an IPv6 server and the IPv4 server will work as it was connected to an IPv4 client.

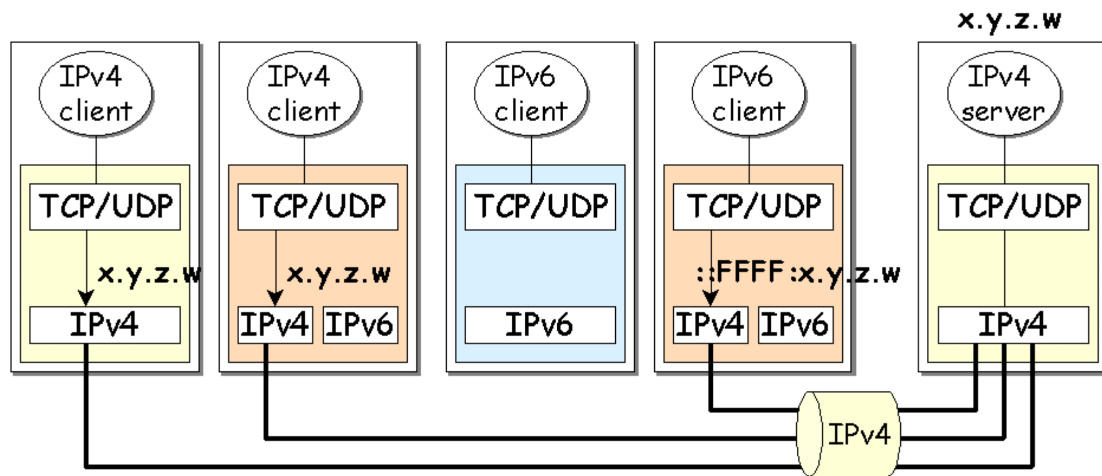

Figure 5: IPv4/IPv6 clients connecting to an IPv4 server at IPv4-only node.

4.2.2 IPv6/IPv4 clients connecting to an IPv6 server at IPv6-only node

There is an IPv4 server application running at IPv6-only node and we will analyze all connection possibilities from clients, see Figure 6.

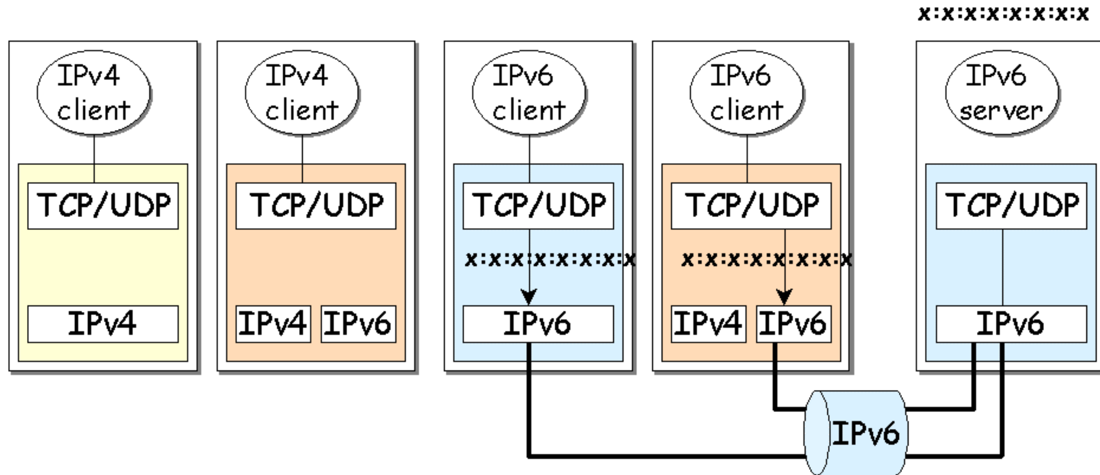

Figure 6: IPv4/IPv6 clients connecting to an IPv6 server at IPv6-only node.

IPv4 clients cannot connect to an IPv6 server running at an IPv6-only node, since IPv4 clients cannot use IPv6 addresses. Then, only IPv6 clients can connect to the server using the IPv6 address of the server node.

4.2.3 IPv6/IPv4 clients connecting to an IPv4 server at dual stack node

There is an IPv4 server application running at dual stack node and we will analyze all connection possibilities from clients, see Figure 7.

When using IPv4 clients, the mechanisms are similar to the 4.1.1 section, the IPv4 server at only-IPv4 node case. Clients use the server IPv4 address. IPv4 packets are exchanged between clients and server.

An IPv6 client at IPv6-only node cannot connect to the IPv4 server, since the server cannot use IPv6 addresses. Although both nodes could communicate using IPv6 protocol, since the server only uses IPv4, the client cannot connect to the server.

An IPv6 client at dual stack node can connect to the IPv4 server using IPv4 network. The IPv6 client request to the resolver the server IP address, the resolver will return the IPv4-mapped IPv6 address to the IPv6 client. As we analyzed at the 4.1.1 section, the IPv6 client will work as it was connected to an IPv6 server and the IPv4 server will work as it was connected to an IPv4 client.

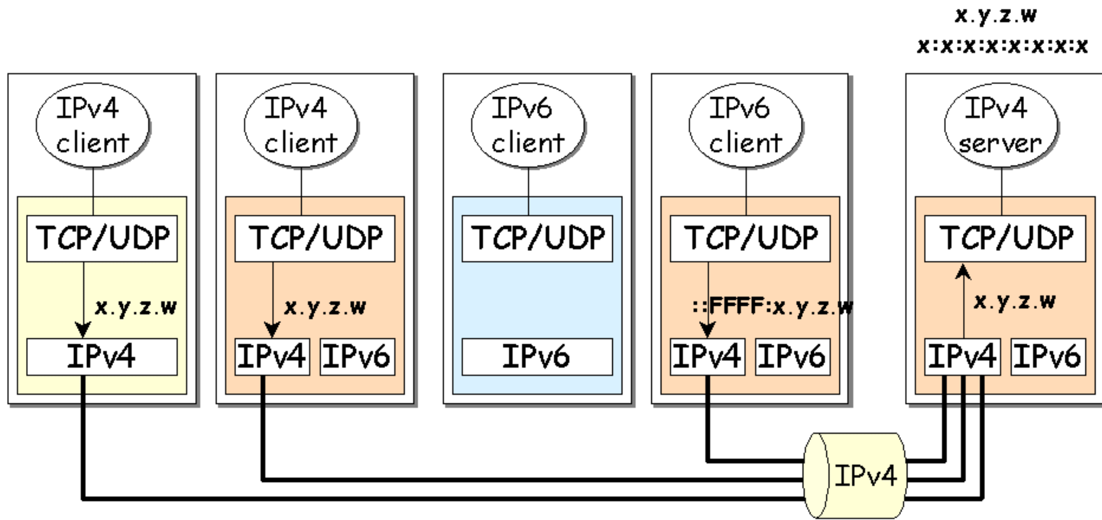

Figure 7: IPv4/IPv6 clients connecting to an IPv4 server at dual stack node.

4.2.4 IPv6/IPv4 clients connecting to an IPv6 server at dual stack node

There is an IPv6 server application running at dual stack node and we will analyze all connection possibilities from clients, see Figure 8.

IPv4 clients will connect using IPv4 protocol. They will use the IPv4 server address and exchange IPv4 packets. These packets are delivered to the IPv6 server using IPv4-mapped IPv6 addresses. Also, the server will use the IPv4-mapped IPv6 addresses when answering the IPv4 client requests. The dual stack at the server node will send IPv4 packets when the IPv4-mapped IPv6 address is used.

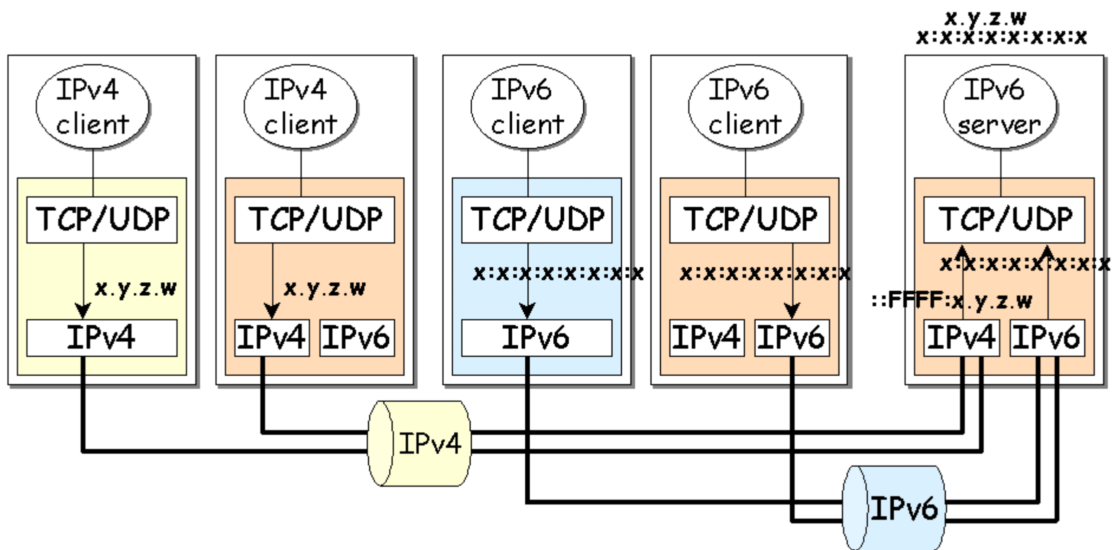

Figure 8: IPv4/IPv6 clients connecting to an IPv6 server at dual stack node.

IPv6 clients will use the server IPv6 address to connect to it and will exchange IPv6 packets.

4.2.5 IPv4/IPv6 clients connecting to an IPv4-only server and IPv6-only server at dual stack node.

During transition there could be running different versions of the same application on a dual stack node, the IPv4-only and the IPv6-only versions. The IPv4-only server will only accept connections from IPv4 clients and the IPv6-only server will only attend to IPv6 clients, see Figure 9.

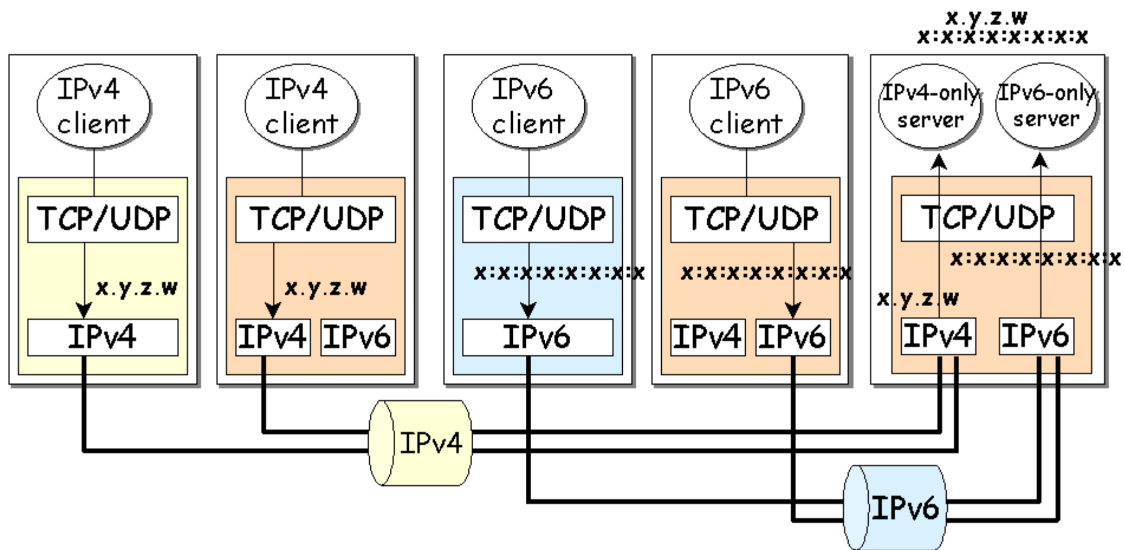

Figure 9: IPv4/IPv6 clients connecting to an IPv4-only server and IPv6-only server at dual stack node.

This case allows separating the IPv4 connections from IPv6 ones. If only IPv6 connections must be accepted, the IPv6-only server will be started. Notice that 4.2.4 is similar to this case, but in that case the IPv6 server will accept all connection requests, from IPv4 and IPv6 clients and cannot be separated connection from different protocol versions.

4.2.6 Client/server and network type interoperability

The Table 2 summarizes the combinations to connect clients and servers running at different kind of nodes: IPv4-only node, IPv6-only node and dual stack node. The combinations signed with “X” denote that communication between such kinds of nodes is not possible. However, dual-stack combinations allow network communication in almost all circumstances. There is only an exception: When the server is IPv4 and an IPv6 client tries to communicate with it, the connection is only possible if client address is an IPv4-mapped into IPv6 address. In this case, if the client chooses a pure IPv6 address, the server will not be able to manage the client address.

Table 2: Client server and network type combinations.

		IPv4 server application		IPv6 server application	
		IPv4 node	Dual-stack	IPv6 node	Dual-stack
IPv4 client	IPv4 node	IPv4	IPv4	X	IPv4
	Dual-stack	IPv4	IPv4	X	IPv4
IPv6 client	IPv6 node	X	X	IPv6	IPv6
	Dual-stack	IPv4	IPv4 / X	IPv6	IPv6

Therefore, four application types can be distinguished:

IPv4-only: An application that is not able to handle IPv6 addresses i.e. it cannot communicate with nodes that do not have an IPv4 address.

IPv6-aware: An application that can communicate with nodes that do not have IPv4 addresses i.e. the application can handle the larger IPv6 addresses. In some cases this might be transparent to the application, for instance when the API hides the content and format of the actual addresses.

IPv6-enabled: An application that, in addition to being IPv6-aware, takes advantage of some IPv6 specific features such as flow labels. The enabled applications can still operate over IPv4, perhaps in a degraded mode.

IPv6-required: An application that requires some IPv6 specific feature and therefore cannot operate over IPv4.

During the gradual transition phase from IPv4 to IPv6, the same application should be run in an IPv4 or IPv6 nodes. Hence, portability is one of the main features of applications, which should work in both environments.

4.3 Porting Source Code

Most existing applications are written assuming IPv4. In general, most of them can be converted without too much effort. Many changes could be done automatically and there are some scripts to do it. However, there are applications that make special use of IPv4 addresses or include advanced features, such as multicasting, IP options or raw sockets. In such cases, exhaustive code revision should be done.

Some changes are needed to adapt the socket API for IPv6 support: a new socket address structure to carry IPv6 addresses, new address conversion functions and several new socket options that are explained in RFC-2553. These extensions are designed to provide access to the basic IPv6 features required by TCP and UDP applications, including multicasting, while introducing a minimum of change into the system and providing complete compatibility for existing IPv4 applications. Access to more advanced features (raw sockets, header configuration, etc.) is addressed in RFC-2292.

The following subsections are an overview of changes in the basic BSD socket API to support IPv6 (see RFC-2553 for details) and how the applications source code should be changed in order to make it portable and protocol independent code.

4.3.1 Socket Address Structures

Functions provided by socket API use socket address structures to determine the communication service access point. Since different protocols can handle socket functions, a generic socket address structure is used as argument of these functions for any of the supported communication protocol families, `sockaddr`.

```
struct sockaddr {
     sa_family_t sa_family;       /* Address family */
     char sa_data[14];            /* protocol-specific address */
};
```

Although socket functions handle generic socket address structure, developers must fill the adequate socket address structure according to the communication protocol they are using to establish the socket. Concretely, the IPv4 sockets use the following structure, `sockaddr_in`:

```
typedef uint32_t in_addr_t;
struct in_addr  {
    in_addr_t s_addr;                        /* IPv4 address */
};

struct sockaddr_in {
     sa_family_t sin_family;                 /* AF_INET */
     in_port_t sin_port;                     /* Port number.  */
     struct in_addr sin_addr;                /* Internet address.  */

     /* Pad to size of `struct sockaddr'.  */
     unsigned char sin_zero[sizeof (struct sockaddr) -
                            sizeof (sa_family_t) -
                            sizeof (in_port_t) -
                            sizeof (struct in_addr)];
};
```

And the IPv6 sockets use the following structure, `sockaddr_in6`, with a new address family `AF_INET6`:

```
struct in6_addr {
    union {
        uint8_t u6_addr8[16];
        uint16_t u6_addr16[8];
        uint32_t u6_addr32[4];
    } in6_u;

    #define s6_addr                 in6_u.u6_addr8
    #define s6_addr16               in6_u.u6_addr16
    #define s6_addr32               in6_u.u6_addr32
```

```
};

struct sockaddr_in6 {
    sa_family_t sin6_family;    /* AF_INET6 */
    in_port_t sin6_port;        /* Transport layer port # */
    uint32_t sin6_flowinfo;     /* IPv6 flow information */
    struct in6_addr sin6_addr;  /* IPv6 address */
    uint32_t sin6_scope_id;     /* IPv6 scope-id */
  };
```

The `sockaddr_in` or `sockaddr_in6` structures are utilized when using respectively IPv4 or IPv6. Existing applications are written assuming IPv4, using `sockaddr_in` structure. They can be easily ported changing this structure by `sockaddr_in6`. However, when writing portable code, it is preferable to eliminate protocol version dependencies from source code. There is a new data structure, `sockaddr_storage`, large enough to store all supported protocol-specific address structures and adequately aligned to be cast to the a specific address structure.

```
/* Structure large enough to hold any socket address (with the
historical exception of AF_UNIX). 128 bytes reserved.  */

#if ULONG_MAX > 0xffffffff
# define __ss_aligntype __uint64_t
#else
# define __ss_aligntype __uint32_t
#endif
#define _SS_SIZE        128
#define _SS_PADSIZE     (_SS_SIZE - (2 * sizeof (__ss_aligntype)))

struct sockaddr_storage
  {
    sa_family_t ss_family;      /* Address family */
    __ss_aligntype __ss_align;  /* Force desired alignment.  */
    char __ss_padding[_SS_PADSIZE];
  };
```

Hence, portable applications should use `sockaddr_storage` structure to store their addresses, IPv4 or IPv6 ones. This new structure hides the specific socket address structure that the application is using.

4.3.2 Socket functions

The socket API has not been changed since it handles generic address structures, independent from the protocol it is using. However, applications should change the value of arguments used to call the socket functions. First, applications should allocate enough memory to store the appropriate socket address structure. And second, before calling the socket functions, the specific socket address structure should be cast to the generic one, which is accepted by the socket functions as an argument.

```
int     socket  (int domain, int type, int protocol);

int     listen  (int s, int backlog);
```

```
ssize_t write   (int fd, const void *buf, size_t count);

int     send    (int s, const void *msg, size_t len, int flags);

int     sendmsg (int s, const struct msghdr *msg, int flags);

ssize_t read    (int fd, void *buf, size_t count);

int     recv    (int s, void *buf, size_t len, int flags);

int     recvmsg (int s, struct msghdr *msg, int flags);

int     close   (int fd);

int     shutdown(int s, int how);
```

Socket calls where a socket address structure is provided from application to kernel.

```
int bind   (int sockfd, struct sockaddr *my_addr, socklen_t addrlen);

int connect(int sockfd, const struct sockaddr *serv_addr,
            socklen_t addrlen);

int sendto (int s, const void *msg, size_t len, int flags,
            const struct sockaddr *to, socklen_t tolen);
```

Socket calls where a socket address structure is provided from kernel to application.

```
int  accept     (int s, struct sockaddr *addr, socklen_t *addrlen);

int  recvfrom   (int s,  void  *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);

int  getpeername(int s, struct sockaddr *name, socklen_t *namelen);

int  getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

4.3.3 Required modifications when porting to IPv6

When porting to IPv6, some modifications related to the socket API are required in the network applications.

Three modification types to be made when porting source code to IPv6 have been identified:

a) Creating a socket.

b) Socket calls where a socket address structure is provided from application to kernel.

c) Socket calls where a socket address structure is provided from kernel to application.

There are some examples below of these three types of operation.

a) Creating a socket

The difference between creating an IPv4 and an IPv6 socket is the value of the family argument in the socket call.

- IPv4 source code:

```
socket(PF_INET, SOCK_STREAM, 0);  /* TCP socket */
socket(PF_INET, SOCK_DGRAM,  0);  /* UDP socket */
```

- IPv6 source code:

```
socket(PF_INET6, SOCK_STREAM, 0); /* TCP socket */
socket(PF_INET6, SOCK_DGRAM,  0); /* UDP socket */
```

b) Socket calls where a socket address structure is provided from application to kernel

Socket address structure is filled before calling the socket function.

- IPv4 source code (a complete example is included in appendix in the ListenServer4.cpp file):

```
struct sockaddr_in addr;
socklen_t          addrlen = sizeof(addr);

/*
   fill addr structure using an IPv4 address before calling socket
   funtion
*/

bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

- IPv6 source code:

```
struct sockaddr_in6 addr;
socklen_t           addrlen = sizeof(addr);

/*
   fill addr structure using an IPv6 address before calling socket
   function
*/

bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

- Portable source code (a complete example is included in appendix, in the ListenServer.cpp file):

```
struct sockaddr_storage addr;
socklen_t               addrlen;

/*
   fill addr structure using an IPv4/IPv6 address and
   fill addrlen before calling socket function
*/

bind(sockfd,(struct sockaddr *)&addr, addrlen);
```

c) Socket calls where a socket address structure is provided from kernel to application

When calling this kind of socket functions, the socket address structure is filled in with the address of the source entity.

- IPv4 source code (a complete example is included in appendix, in the TCPDayTimeServer4.cpp file):

```
struct sockaddr_in addr;
socklen_t          addrlen = sizeof(addr);

accept(sockfd,(struct sockaddr *)&addr, &addrlen);

/*
   addr structure contains an IPv4 address
*/
```

- IPv6 source code:

```
struct sockaddr_in6 addr;
socklen_t           addrlen = sizeof(addr);

accept(sockfd,(struct sockaddr *)&addr, &addrlen);

/*
   addr structure contains an IPv4 address
*/
```

- Portable source code (a complete example is included in appendix, in the TCPDayTimeServer.cpp file):

```
struct sockaddr_storage addr;
socklen_t               addrlen = sizeof(addr);

accept(sockfd,(struct sockaddr *)&addr, &addrlen);

/*
   addr structure contains an IPv4/IPv6 address
   addrlen contains the size of the addr structure returned
 */
```

4.3.4 Address conversion functions

The address conversion functions convert between binary and text address representation. Binary representation is the network byte ordered binary value, which is stored in the socket address structure and the text representation, named presentation, is an ASCII string.

The IPv4 address conversion functions are the following ones:

```
/*
   From text to IPv4 binary representation
*/
int       inet_aton (const char *cp, struct in_addr *inp);
in_addr_t inet_addr( const char *cp);

/*
   From IPv4 binary to text representation
*/
char     *inet_ntoa(struct in_addr in);
```

The new address conversion functions which work with both IPv4 and IPv6 addresses are the following ones:

```
/*
   From presentation to IPv4/IPv6 binary representation
*/
int inet_pton(int family, const char *src, void *dst);

/*
   From IPv4/IPv6 binary to presentation
*/
const char *inet_ntop(int family, const void *src,
                      char *dst, size_t cnt);
```

- IPv4 source code (a complete example is included in appendix, in the TCPDayTimeServer.cpp file):

```
struct sockaddr_in addr;
char *straddr;

memset(&addr1, 0, sizeof(addr));
addr.sin_family = AF_INET;        /* family */
addr.sin_port = htons(MYPORT);    /* port, networt byte order */

/*
   from text to binary representation
*/
inet_aton("138.4.2.10", &(addr.sin_addr));


/*
   from binary to text representation
*/
straddr = inet_ntoa(addr.sin_addr);
```

- IPv6 source code:

```
struct sockaddr_in6 addr;
char straddr[INET6_ADDRSTRLEN];

memset(&addr, 0, sizeof(addr));
addr.sin6_family = AF_INET6;       /* family */
addr.sin6_port = htons(MYPORT);    /* port, networt byte order */

/*
   from presentation to binary representation
*/
inet_pton(AF_INET6, "2001:720:1500:1::a100",
          &(addr.sin6_addr));

/*
   from binary representation to presentation
*/
inet_ntop(AF_INET6, &addr.sin6_addr, straddr,
          sizeof(straddr));
```

4.3.5 Resolving names

Applications should use names instead of addresses for hosts. Names are easier to remember and remain the same; however numeric addresses could change more frequently.

From applications point of view the name resolution is a system-independent process. Applications call functions in a system library known as the resolver, typically `gethostbyname` and `gethostbyaddr`, which is linked into the application when the application is built. The resolver code is the burden of making the resolution dependent of the system configuration.

There are two new functions to make name and address conversions protocol independent, `getaddrinfo` and `getnameinfo`. Besides, the use of these new ones instead of `gethostbyname` and `gethostbyaddr` is recommended because the latter are not normally reentrant and could provoke problems in threaded applications.

The `getaddrinfo` function returns a linked list of `addrinfo` structures, which contains information requested for a specific set of hostname, service and additional information stored in an `addrinfo` structure.

```
struct addrinfo {
    int     ai_flags;            /* AI_PASSIVE, AI_CANONNAME */
    int     ai_family;           /* AF_UNSPEC, AF_INET, AF_INET6 */
    int     ai_socktype;         /* SOCK_STREAM, SOCK_DGRAM ... */
    int     ai_protocol;         /* IPPROTO_IP, IPPROTO_IPV6 */
    size_t  ai_addrlen;          /* length of ai_addr */
    struct sockaddr ai_addr;     /* socket address structure */
    char   ai_canonname;         /* cannonical name */
    struct addrinfo ai_next;     /* next addrinfo structure */
};
```

```
/* function to get socket address structures */

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

When writing a typical client application, node and service are normally specified. When writing a server application, they both can be specified too, but in many cases only service is specified, allowing clients to connect to any node interfaces.

Applications should examine the linked list returned by `getaddrinfo` to use the adequate structure. In some cases, not all the addresses returned by this function can be used to create a socket.

The `getaddrinfo` function allocates a set of resources for the returned linked list. The `freeaddrinfo` function frees these resources.

```
/* function to free the resources allocated by getaddrinfo */

void freeaddrinfo(struct addrinfo *res);
```

Next, an example of `getaddrinfo` and `freeaddrinfo` usage (complete examples are included in appendix, in the listenServer.cpp and connectClient.cpp files):

```
n = getaddrinfo(hostname, service, &hints, &res);

/*
    Try open socket with each address getaddrinfo returned,
    until getting a valid socket.
*/

resave = res;

while (res) {
    sockfd = socket(res->ai_family,
                    res->ai_socktype,
                    res->ai_protocol);

    if (!(sockfd < 0))
        break;

    res = res->ai_next;
}

freeaddrinfo(ressave);
```

The `getnameinfo` function provides from a socket address structure, the address and service as character strings. Next, an example of use (a complete example is included in appendix, in the UDPDayTimeServer.cpp file):

```
char clienthost   [NI_MAXHOST];
char clientservice[NI_MAXSERV];

/* ... */

/* listenfd is a server socket descriptor waiting connections
   from clients
*/

connfd = accept(listenfd,
                (struct sockaddr *)&clientaddr,
                &addrlen);

getnameinfo((struct sockaddr *)&clientaddr, addrlen,
            clienthost, sizeof(clienthost),
            clientservice, sizeof(clientservice),
            NI_NUMERICHOST);

printf("Received request from host=[%s] port=[%s]\n",
       clienthost, clientservice);
```

Typically, applications do not require knowing the version of the IP they are using. Hence, applications only should try to establish the communication using each address returned by resolver until it works. However, applications could have a different behaviour when using IPv4, IPv6, IPv4-compatible-IPv6 or IPv4-mapped, etc. addresses.

There are defined some macros to help applications to test the type of address they are using, see Table 3.

Table 3: Macros for testing type of addresses.

```
int  IN6_IS_ADDR_UNSPECIFIED    (const struct in6_addr *);
int  IN6_IS_ADDR_LOOPBACK       (const struct in6_addr *);
int  IN6_IS_ADDR_MULTICAST      (const struct in6_addr *);
int  IN6_IS_ADDR_LINKLOCAL      (const struct in6_addr *);
int  IN6_IS_ADDR_SITELOCAL      (const struct in6_addr *);
int  IN6_IS_ADDR_V4MAPPED       (const struct in6_addr *);
int  IN6_IS_ADDR_V4COMPAT       (const struct in6_addr *);
int  IN6_IS_ADDR_MC_NODELOCAL   (const struct in6_addr *);
int  IN6_IS_ADDR_MC_LINKLOCAL   (const struct in6_addr *);
int  IN6_IS_ADDR_MC_SITELOCAL   (const struct in6_addr *);
int  IN6_IS_ADDR_MC_ORGLOCAL    (const struct in6_addr *);
int  IN6_IS_ADDR_MC_GLOBAL      (const struct in6_addr *);
```

4.3.6 Multicasting

When using UDP multicast facilities some changes must be carried out to support IPv6. First application must change the multicast IPv4 addresses to the IPv6 ones, and second, the socket configuration options.

IPv6 multicast addresses begin with the following two octets: FF0X.

The multicast socket options are used to configure some of parameters for sending multicast packets, see Table 4.

Applications using multicast communication open a socket and need to configure it to receive multicast packets. This is the main difference between the multicast and unicast socket use. Once the socket is opened and a multicast address is bind, it is required to configure at least the membership option to join to the multicast group. It will allow receiving all information sent to this multicast group.

Table 4: Multicast socket options.

IPv6 OPTION	
IPV6_MULTICAST_IF	Interface to use for outgoing multicast packets.
IPV6_MULTICAST_HOPS	Hop limit for multicast packets.
IPV6_MULTICAST_LOOP	Multicast packets are looped back to the local application.
IPV6_ADD_MEMBERSHIP	Join a multicast group.
IPV6_DROP_MEMBERSHIP	Leave a multicast group.

In the following example it is shown how can be configured, in both cases with IPv4 and IPv6, three multicast socket options: LOOP, MEMBERSHIP and TTL. The same function, `setsockopt`, is used in both cases. The main difference is the option constant values and the multicast group addresses. In IPv4 the multicast group is represented using a `sockaddr_in` structure and in IPv6 it is represented using a `sockaddr_in6`.

```
int
joinGroup(int sockfd, int loopBack, int mcastTTL,
          struct sockaddr_storage *addr)
{
    int r1, r2, r3, retval;

    retval=-1;

    switch (addr->ss_family) {
        case AF_INET: {
            struct ip_mreq      mreq;

            mreq.imr_multiaddr.s_addr=
                ((struct sockaddr_in *)addr)->sin_addr.s_addr;
            mreq.imr_interface.s_addr= INADDR_ANY;

            r1= setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
                           &loopBack, sizeof(loopBack));
            if (r1<0)
                perror("joinGroup:: IP_MULTICAST_LOOP:: ");

            r2= setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_TTL,
                           &mcastTTL, sizeof(mcastTTL));
            if (r2<0)
               perror("joinGroup:: IP_MULTICAST_TTL:: ");

            r3= setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                           (const void *)&mreq, sizeof(mreq));
            if (r3<0)
                perror("joinGroup:: IP_ADD_MEMBERSHIP:: ");

        } break;

        case AF_INET6: {
           struct ipv6_mreq    mreq6;

           memcpy(&mreq6.ipv6mr_multiaddr,
                  &(((struct sockaddr_in6 *)addr)->sin6_addr),
                  sizeof(struct in6_addr));

           mreq6.ipv6mr_interface= 0; // cualquier interfaz

           r1= setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
                          &loopBack, sizeof(loopBack));
           if (r1<0)
               perror("joinGroup:: IPV6_MULTICAST_LOOP:: ");

           r2= setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
                          &mcastTTL, sizeof(mcastTTL));
```

```
            if (r2<0)
                perror("joinGroup:: IPV6_MULTICAST_HOPS::  ");

            r3= setsockopt(sockfd, IPPROTO_IPV6,
                           IPV6_ADD_MEMBERSHIP, &mreq6, sizeof(mreq6));
            if (r3<0)
               perror("joinGroup:: IPV6_ADD_MEMBERSHIP:: ");

        } break;

        default:
            r1=r2=r3=-1;
    }

    if ((r1>=0) && (r2>=0) && (r3>=0))
        retval=0;

    return retval;
}
```

The complete description of multicast daytime example can be seen in appendix.

5. Guidelines on developing new applications

In the design of applications to use IPv6 some characteristics must be taken into account. First of all it is necessary to separate the transport module from the rest of application functional modules. This separation makes the application independent on the network system used. Then, if the network protocol is changed, only the transport module should be modified. Transport module should provide the communication channel abstraction with basic channel operations and generic data structures to represent the addresses. These abstractions could be instantiated as different implementations depending on the network protocol required at any moment. The application will deal with this generic communication channel interface without knowing the network protocol used. Using this design if a new network protocol is added, application developers only need to implement a new instance of the channel abstraction, which manages the features of this new protocol.

Once the transport module has been designed, there are some implementations details related to the type of the nodes which will run the application: IPv4-only nodes, IPv6-only nodes or both, dual stack.

Within the transport module the use of the new API with extensions for IPv6 is mandatory, but it is strongly recommended to make the program protocol independent (for instance, using the BSD socket API consider getaddrinfo and getnameinfo instead of gethostbyname and gethostbyaddr). The new IPv6 functions are only valid if this protocol is supported by all installed systems. IPv6 is now reaching maturity and most popular systems provide it as default in their standard distributions. However, IPv6 support does not force to use it, only after the complete network configuration is defined, applications will use IPv4 or IPv6.

Protocol independent code is feasible if the design is based on the principal building block for transitioning, the dual stack. Dual stacks maintain two protocol stacks that operate in parallel and thus it is allowed to operate via either protocol. The operating system running dual stack translates IPv4 addresses to IPv4-mapped IPv6 ones when communicating to IPv4 remote applications.

Although applications are written following program protocol independent rules, other points have to be considered such as the movement of binary code between IPv4-only nodes, dual stacks nodes or IPv6-only nodes.

Compilation options (`#ifdefs` in C language) can be provided throughout the code to select the proper use environment. If IPv6 is not supported the IPv4-only code will be selected for compilation during installation process. However, if IPv6 is supported (Kernel level support) by installed systems, the code for IPv6 or the IPv4 could be selected for compilation, depending on the requirements of applications. Notice that if the kernel supports IPv6, it only means the IPv6 option could be activated and while this option is disabled the node will be only use IPv4 stack. During the transition period nodes are usually running dual stack, both IPv4 and IPv6 stacks. The problem with the conditional compilation approach is that the code becomes littered with compilation options very quickly and harder to follow and maintain.

If an application is compiled on a system, which supports dual stack and move the binary code to an IPv4-only node without IPv6 kernel support, the source code must be recompiled to use the original IPv4 API. The binary code generated on the dual stack uses the new system functions, which are not supported in the IPv4-only node.

If the binary code is moved, which has been compiled on a dual stack, to an IPv4-only node with IPv6 kernel support and IPv6 stack not activated, recompilation is not required. Since the IPv6 stack is not activated, the node cannot establish IPv6 connections. However, the resolver system could return an IPv6 address to an application query and the application should be prepared to discard this IPv6 address and select the IPv4 one to open connections.

All these alternatives are summarized in Table 5.

Table 5: IPv4 or IPv6 activation.

<table>
<tr><th colspan="2">NODE</th><th colspan="2">APPLICATION</th><th colspan="2">CONNECTIONS</th></tr>
<tr><th>IPv6 kernel support</th><th>Dual stack activated</th><th>Network API used</th><th>Application type</th><th>IPv4 connection</th><th>IPv6 connection</th></tr>
<tr><td rowspan="2">Yes</td><td>Yes</td><td>IPv6 extensions (portable code)</td><td>IPv6-enabled</td><td>IPv4 stack</td><td>IPv6 stack</td></tr>
<tr><td>No</td><td>IPv6 extensions (portable code)</td><td>IPv6-enabled</td><td>IPv4 stack</td><td>IPv6 address resolution
Connection error.</td></tr>
<tr><td colspan="2">No</td><td>Without IPv6 extensions
(old API)</td><td>IPv4-only (compiled with IPv4 options)</td><td>IPv4 stack</td><td>Error</td></tr>
</table>

In summary, if applications follow the recommendations explained above, using a separated protocol independent transport module, which provides a generic communication API, it is easy to adapt them to new network protocols. Besides, the generic communication API could be implemented as a communication library to be used for many applications. This solution encourages the code reusability and makes communication modules of applications easy to maintain.

Finally, if the new application is designed only for the new IPv6 environment, the code is simpler than dual stack implementation. The new application will work only with other IPv6 nodes. When the local IPv6 node wants to use IPv6 application to communicate with a remote IPv4 node, one standard network transition mechanism should be provided [NAT-PT] [SIIT] [6to4]. Depending on the application, sometimes it is possible to use special dual-stack edge servers at the IPv4 and IPv6 border as application level proxies, removing the need for network-level transition.

6. Conclusions

Today, IP has established itself as a primary vehicle for our global system of electronic communication enabling a vast array of client/server computing applications. Although the IP success story took years to unfold, it's the time to make refinement plans for the current IP version (IPv4). The IP next generation known as IPng or IPv6 is both a near-term and long-range concern for network owners and service providers. Though it is based on much-needed enhancements to IPv4 standards, IPv6 should by viewed as a broad retooling project that will ultimately provide a much-needed evolutionary architecture of today's applications and networks.

Transition between today's IPv4 Internet and the new IPv6 one will be a long process during which both protocol versions will coexist. But transition is not only related to networks, but also to involved applications. Existing applications are written assuming IPv4 and it takes a long time to produce new versions to work over IPv6 interface.

The simplest transition scenario combines dual stack network with a mechanism to transform all IPv6 related operations to the traditional well-known transport interface defined for IPv4 [BIS, BIA]. These techniques maintain actual applications (IPv4) and introduce IPv6 only at network level. They are based on the insertion of translation modules devoted to intercept and transform network data between application and network and vice-versa.

Unless all previous procedures are valid in a very early phase, finally applications should be ported to the new scenario. Sometimes IPv6 version can be developed in parallel with other programmed modifications. However, in most cases we only want to maintain the initial functionality and adapt communications module to enable IPv6.

There are two probable scenarios during transition. The first is based on maintaining two application versions and the second with only one dual version.

We can forget old IPv4 applications and maintain them only to communicate with IPv4 nodes during transition period. The transition methodology consists in developing a complete set of applications designed to work only over IPv6 transport layer. The main advantage is to be ready with new applications after transition period. To finish IPv6 transition it is necessary only to removed IPv4 applications and dual stack network. However, the selection of suitable applications during transition period it's a user responsibility. When tests, configuration or management of applications are considered, the application selection is not a problem, because their users normally have technical knowledge. However, with most of end-user applications the user hasn't enough knowledge or doesn't want to know which is the proper application version should be used in each connection. Moreover, during transition period will be necessary to provide data servers working both IPv4 and IPv6 to accept connections not only from new but also from old nodes.

The second transition scenario looks for changing existing applications, not developing new ones. The porting process takes the original IPv4 application, review source code and produces a new version adapted to work over both IPv4 and IPv6 environments. The porting process is a little bit more complex than IPv6 only support, but the result is more flexible to be used during transition period. Transition period will be probably a long period during which there will be islands, not only IPv6 but also IPv4, interested for users in the new environments. This is because during a long period most of data servers will be maintained in the IPv4 only environment.

Reviewing existing applications to work both IPv4 and IPv6 simultaneously requires two phases: code analysis and code rewriting. Code examination is required to locate the following issues:

- IP addresses management
- Network information storage: data structures
- Communications API functions and pre-defined constants

After code identification, code-rewriting can take place. Many changes can be done automatically and there are some applications to do it [SUN, HP]. However, many applications make special use of IPv4 addresses and socket structures, consequently automatic rewriting is not possible. The most important issues that should be taken into account are the following:

- Names resolution,
- Sockets address structures,
- Sockets programming interface,
- Address conversion functions and
- Multicast interface

Finally, if a design of a new IPv6 application is considered, some recommendations should be taken into account. First of all, it should be decided if the new application will be ready for IPv6 only or it will be a dual one. IPv6 only application doesn't know IPv4 protocol, therefore v6 to v4 adaptation should be added to the system when IPv4 interaction is demanded. However, a dual application is a little more complex than previous one.

Applications should include code to select the proper interface depending on the correspondent node. To isolate all these operations from the rest of the application a transport module is recommended. The transport module provides a clear and uniform interface to the rest of the application hiding all protocol selection details.

This complex scenario can be summarized in the following items:

- Service continuity: IPv6 transition is not only an address or routing issue but also mainly a service enhancement. All commercial services deployed recently such as QoS, Intranets, group collaboration or IP telephony, have to be continuously provided whatever the IP infrastructure might be.
- Mixed scenario during transition: The introduction of IPv6 will be slow compared with the size of Internet. When IPv4 and IPv6 have to coexist, keeping transition under control is essential to avoid a final scenario with two parallel Internet infrastructures. Therefore, the application porting process should be included with enough resources in the transition plan. In standard environments, the application porting process has a reduced cost because most of typical applications are already IPv6 enabled. The LONG Project has made an updated catalogue of such applications.
- Transition is not always a necessary solution: Deploying new applications not demanding a complete development to support IPv4 in combination with IPv6 users. Next generation applications will be available only in the new environment.

Deploying transition mechanisms at a large scale can lead to scalability issues that could heavily limit the IPv6 performance compared to a native solution. Moreover, the availability of new applications only in the new environment is the best mechanism to accelerate transition from actual IPv4 environment to the new IPv6 one.

7. Bibliography

[APPT] Myung-Ki Shin et al, *Application Aspects of IPv6 Transition*. <draft-shin-ngtrans-application-transition-00.txt>, May 2002.

[ASI] W. Stevens, M. Thomas. *Advanced Sockets API for IPv6*. February 1998. RFC2292.

[ASAPI] W. Richard Stevens, *Advanced Sockets API for IPv6*. <draft-ietf-ipngwg-rfc2292bis-07.txt> Expires: October 19, 2002. Obsoletes RFC 2292.

[AURL] R. Hinden, B., Carpenter, L. Masinter, *Format for Literal IPv6 Addresses in URL's*. RFC2732. December 1999.

[BIA] Seungyun Lee et al, *Dual Stack Hosts using Bump–In-The-API (BIA)*. <draft-ietf-ngtrans-bia-01.txt>, November 2001, work in progress.

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

[BISE] R.E. Gilligan, *Basic Socket Interface Extensions for IPv6*. <draft-ietf-ipngwg-rfc2553bis-05>. Obsoletes RFC 2553. February 2002.

[BSI] R. Gilligan, S., Thomson, J. Bound, W. Steven. *Basic Socket Interface Extensions for IPv6*. March 1999. RFC2553.

[DASL] Richard Draves, *Default Address Selection for IPv6*. <draft-ietf-ipv6-default-addr-select-08.txt>. June 17, 2002.

[DSF] K. Nicholsms, S. Blake, F. Baker, D. Black, *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*. RFC2474.

[DSTM] Jim Bound et al, *Dual Stack Transition Mechanism (DSTM)*. <draft-ietf-ngtrans-dstm-08.txt>, December 2002.

[DYTP] J. Postel. *Daytime Protocol*. RFC867. May 1983.

[HPMT] K S Gundu Rao and Ramesh, HP - *UX IPv4 - IPv6 migration tool*. November 2002.

[IPV6] S. Deering and R. Hinden. *Internet Protocol, Version 6 (IPv6) Architecture*. RFC 2460, December 1998.

[LD31] T. de Miguel, E. Castro et al, *Requirements and guidelines for distributed laboratorios applications migration*. LONG Project IST-1999-20393, D3.1. May 2001.

[MTUD] J. McCann, S. Deering, J. Mogul, *Path MTU Discovery for IP version 6*. RFC1981. August 1996.

[NAT] Srisuresh, P. and K. Egevang, *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022, January 2001.

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

[PMTU] K. Lahey, *TCP problems with Path MTU Discovery,*. September 2000.

[PAPPS] *Porting Applications to the IPv6 APIs*, Solaris Version 8. Sun Microsystems Inc. Revision 2. October 1999.

[RRNM] M. Crawford, *Router Renumbering for IPv6*. RFC2894. August 2000.

[SIIT] Nordmark, E., *Stateless IP/ICMP Translator (SIIT),* RFC 2765, February 2000.

[STV1] Richard Stevens. *The Unix Network Programming* – Volume 1.

[STV2] Richard Stevens, Gary Wright. *TCP/IP Illustrated* – Volume 2.

[TMHR] R Gilligan and E Nordmark*, Transition mechanisms for IPv6 Hosts and Routers.* RFC 2893.[V6AA] R. Hinden, S. Deering, IP Version 6 Addressing Architecture. RFC2373. July 1998.

8. Appendix: Examples of real porting process

This appendix is devoted to show some simple applications, which can be used to illustrate porting procedures. This appendix is used to review characteristics and methods described in previous sections.

Porting guidelines are valid for any programming language, however for simplicity, application-porting examples are provided only in C language. All these examples have been tested in a SuSE Linux 8.0 distribution, kernel version 2.4.10.

8.1 Original daytime version

Daytime is a simple utility, which returns the time and date of a node in a human-readable format. It is defined in RFC 867 and it works at the port number 13.

It is a good example to show porting guidelines with a very simple application. Like most of distributed applications, daytime is composed of a client and a server programs. It can operate over TCP and UDP.

8.1.1 IPv4 Daytime server (TCP/UDP)

The server is composed of a main program and a general function `listen_server` to create and configure a server socket. First, we will analyze this function which will be used by the daytime server program, both TCP and UDP versions.

The `listen_socket` function has the following parameters: `hostname` to wait connections in a specific network interface, `service` to determine the port number (in this case, the family must be `AF_INET to use IPv4` protocol) and `socktype` to determine if using TCP or UDP protocol.

The listenServer.h file contains the `listen_server` function definition and the listenServer.cpp contains its implementation.

File "ListenServer4.h"

```
#ifndef listen__server__h__
#define listen__server__h__

/*
    listen_server
        creates a server server socket listening at a hostname:
        service using the socket type specified in
        the function arguments.
*/

int
listen_server(const char *hostname,
              const char *service,
              int         socktype);

#endif
```

File "ListenServer4.cpp"

The `listen_socket` function requests socket address information to the resolver module. From this information the `listen_socket` tries to create a new socket, configures it to accept incoming connection requests from clients and returns the valid socket descriptor value.

```
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>

#include "listenServer.h"

const int LISTEN_QUEUE=128;

int
listen_server(const char *hostname,
              const char *service,
              int         socktype)
{

    struct sockaddr_in sin;
    struct hostent  *phe;
    struct servent  *pse;
    struct protoent *ppe;
    int sockfd;
    char *protocol;

    memset(&sin, 0, sizeof(sin));

    sin.sin_family=AF_INET;

    switch(socktype) {

        case SOCK_DGRAM:
            protocol= "udp";
            break;

        case SOCK_STREAM:
            protocol= "tcp";
            break;

        default:
            fprintf(stderr,
                    "listen_server:: unknown socket type=[%d]\n",
                    socktype);
            return -1;
    }


    if (( pse = getservbyname(service, protocol) )) {
        sin.sin_port = pse->s_port;
```

```
    } else if ( (sin.sin_port = htons((u_short)atoi(service))) ==0) {
        fprintf(stderr,
                "listen_server:: could not get service=[%s]\n",
                service);
        return -1;
    }

    if (!hostname) {
        sin.sin_addr.s_addr= INADDR_ANY;

    } else {
        if ((phe = gethostbyname(hostname))) {
            memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);

        } else if ( (sin.sin_addr.s_addr = inet_addr(hostname)) ==
                    INADDR_NONE) {
             fprintf(stderr,
                     "listen_server:: could not get host=[%s]\n",
                     hostname);
             return -1;
        }
    }

    if ((ppe = getprotobyname(protocol)) == 0) {
         fprintf(stderr,
                 "listen_server:: could not get protocol=[%s]\n",
                 protocol);
         return -1;
    }

    if ((sockfd = socket(PF_INET, socktype, ppe->p_proto)) < 0) {
        fprintf(stderr,
                "listen_server:: could not open socket\n");
        return -1;
    }

    if (bind(sockfd, (struct sockaddr *)&sin, sizeof(sin)) != 0) {
        fprintf(stderr,
                "listen_server:: could not bind socket\n");
        close(sockfd);
        return -1;
    }

    listen(sockfd, LISTEN_QUEUE);

    return sockfd;
}
```

From the `listen_socket` function we can build the dayTime server program.

File "TCPDayTimeServer4.cpp"

The TCP dayTime server creates a TCP server socket using the `listen_socket` function and waits for TCP client connections.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>

#include "listenServer.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int listenfd, connfd, port;
    socklen_t addrlen;
    char timeStr[256];
    char *clienthost;
    struct sockaddr_in clientaddr;
    time_t now;

    /* local server socket listening at daytime port=13 */
    listenfd = listen_server(NULL,DAYTIME_PORT, SOCK_STREAM);


    if (listenfd < 0) {
         fprintf(stderr,
                 "listen_socket error:: could not create listening "
                 "socket\n");
         return -1;
    }

    for ( ; ;) {
        addrlen = sizeof(clientaddr);

        /* accept daytime client connections */
        connfd = accept(listenfd,
                        (struct sockaddr *)&clientaddr,
                        &addrlen);

        if (connfd < 0)
            continue;

        clienthost = inet_ntoa(clientaddr.sin_addr);
        port = ntohs(clientaddr.sin_port);
        printf("Received request from host=[%s] port=[%d]\n",
               clienthost, port);
```

```
        /* process daytime request from a client */
        memset(timeStr, 0, sizeof(timeStr));
        time(&now);
        sprintf(timeStr, "%s", ctime(&now));
        write(connfd, timeStr, strlen(timeStr));
        close(connfd);
    }

    return 0;
}
```

File "UDPDayTimeServer4.cpp"

The UDP dayTime server creates an UDP server socket using the `listen_socket` function and waits for UDP client connections.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <errno.h>

#include "listenServer.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int listenfd, connfd, port, n;
    socklen_t addrlen;
    char timeStr[256];
    char request[256];
    char *clienthost;
    struct sockaddr_in clientaddr;
    time_t now;

    /* local server socket listening at daytime port=13 */
    listenfd = listen_server(NULL,DAYTIME_PORT, SOCK_DGRAM);


    if (listenfd < 0) {
         fprintf(stderr,
                 "listen_socket error:: could not create listening "
                 "socket\n");
         return -1;
    }

    addrlen = sizeof(clientaddr);

    for ( ; ;) {
        /* accept daytime client connections */
```

```
        n = recvfrom(listenfd,
                     request,
                     sizeof(request),
                     0,
                     (struct sockaddr *)&clientaddr,
                     &addrlen);

        if (connfd < 0)
            continue;

        clienthost = inet_ntoa(clientaddr.sin_addr);
        port = ntohs(clientaddr.sin_port);
        printf("Received request from host=[%s] port=[%d]\n",
               clienthost, port);

        /* process daytime request from a client */
        memset(timeStr, 0, sizeof(timeStr));
        time(&now);
        sprintf(timeStr, "%s", ctime(&now));

        n = sendto(listenfd, timeStr, sizeof(timeStr), 0,
                   (struct sockaddr *)&clientaddr,
                   addrlen);
    }

    return 0;
}
```

8.1.2 IPv4 Daytime client (TCP/UDP)

The daytime service is reduced to a request from the client and an answer from the server. In TCP version, the request is combined with connection a request. Therefore, the protocol answer is also combined with a connection acceptance message.

The daytime client is structured in two parts: connection phase and operation phase. Connection phase is grouped in one function (`connect_client`) and operation phase is reduced to a single `read` (or a `recvfrom` in the UDP version).

File "ConnectClient4.h"

```
#ifndef connect__client__h__
#define connect__client__h__

int
connect_client (const char *hostname,
                const char *service,
                int         socktype);
#endif
```

File "ConnectClient4.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include "connectClient.h"

int
connect_client (const char *hostname,
                const char *service,
                int         socktype)
{
    struct sockaddr_in sin;
    struct hostent  *phe;
    struct servent  *pse;
    struct protoent *ppe;
    int sockfd;
    char *protocol;

    memset(&sin, 0, sizeof(sin));

    sin.sin_family=AF_INET;

    switch(socktype) {

        case SOCK_DGRAM:
            protocol= "udp";
            break;

        case SOCK_STREAM:
            protocol= "tcp";
            break;

        default:
            fprintf(stderr,
                    "listen_server:: unknown socket type=[%d]\n",
                    socktype);
            return -1;
    }

    if (( pse = getservbyname(service, protocol) )) {
        sin.sin_port = pse->s_port;

    } else if ((sin.sin_port = htons((u_short)atoi(service)))==0) {
         fprintf(stderr,
                 "connec_client:: could not get service=[%s]\n",
                 service);
         return -1;
    }

    if (!hostname) {
        fprintf(stderr,
                "connect_client:: there should be a hostname!\n");
        return -1;

    } else {
        if ((phe = gethostbyname(hostname))) {
```

```
            memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);

        } else if ( (sin.sin_addr.s_addr = inet_addr(hostname)) ==
                                INADDR_NONE) {
             fprintf(stderr,
                     "connect_client:: could not get host=[%s]\n",
                     hostname);
             return -1;
        }
    }

    if ((ppe = getprotobyname(protocol)) == 0) {
         fprintf(stderr,
                 "connect_client:: could not get protocol=[%s]\n",
                 protocol);
         return -1;
    }

    if ((sockfd = socket(PF_INET, socktype, ppe->p_proto)) < 0) {
        fprintf(stderr,
                "connect_client:: could not open socket\n");
        return -1;
    }

    if (connect(sockfd,(struct sockaddr *)&sin, sizeof(sin)) < 0) {
        fprintf(stderr,
                "connect_client:: could not connect to host=[%s]\n",
                hostname);
        return -1;
    }

    return sockfd;
}
```

File "TCPDayTimeClient4.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#include "connectClient.h"

const char *DAYTIME_PORT="13";


int
main(int argc, char *argv[])
{
    int connfd;
    char *myhost;
    char timeStr[256];

    myhost = "127.0.0.1";
```

```
    if (argc > 1)
        myhost = argv[1];

    connfd = connect_client(myhost,
                            DAYTIME_PORT,
                            SOCK_STREAM);

    if (connfd < 0) {
         fprintf(stderr,
           "client error:: could not create connected socket\n");
         return -1;
    }

    memset(timeStr, 0, sizeof(timeStr));

    while (read(connfd, timeStr, sizeof(timeStr)) > 0)
        printf("%s", timeStr);

    close(connfd);

    return 0;
}
```

File "UDPDayTimeClient4.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#include "connectClient.h"

const char *DAYTIME_PORT="13";


int
main(int argc, char *argv[])
{
    int connfd;
    char *myhost;
    char timeStr[256];

    myhost = "127.0.0.1";
    if (argc > 1)
        myhost = argv[1];

    connfd = connect_client(myhost,
                            DAYTIME_PORT,
                            SOCK_STREAM);

    if (connfd < 0) {
         fprintf(stderr,
           "client error:: could not create connected socket\n");
         return -1;
```

```
    }

    letter = '1';
    m= write(connfd, &letter, sizeof(letter));

    memset(timeStr, 0, sizeof(timeStr));

    read(connfd, timeStr, sizeof(timeStr));
    printf("%s\n", timeStr);

    close(connfd);

    return 0;
}
```

8.2 The unicast daytime ported to IPv6

In the following sections it is shown how daytime service is ported from IPv4 to IPv6. Following previous described porting guidelines and after previous program analysis, the code should be reviewed to support IPv4 and IPv6 without compilation.

IPv6 sockets library is similar to IPv4 version and therefore server or client code does not change for the operation phase. The connection phase maintains the same connection model, however the code to set up connections should be adapted.

8.2.1 Daytime server (TCP/UDP)

The TCP and UDP versions of the daytime server program use a common function to create the server socket, the `listen_server` function. This function generates a server socket from a hostname, the service, socket family (IPv4 or IPv6) and socket type (TCP or UDP) parameters.

File "listenServer.h"

```
#ifndef listen__server__h__
#define listen__server__h__

/*
    listen_server
        creates a server server socket listening at a hostname:service
        using the family and socket type specified in the function
        arguments.
*/

int
listen_server(const char *hostname,
              const char *service,
              int         family,
              int         socktype);

#endif
```

File "listenServer.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "listenServer.h"

const int LISTEN_QUEUE=128;

int
listen_server(const char *hostname,
              const char *service,
              int         family,
              int         socktype)
{
    struct addrinfo hints, *res, *ressave;
    int n, sockfd;

    memset(&hints, 0, sizeof(struct addrinfo));

    /*
       AI_PASSIVE flag: the resulting address is used to bind
       to a socket for accepting incoming connections.
       So, when the hostname==NULL, getaddrinfo function will
       return one entry per allowed protocol family containing
       the unspecified address for that family.
    */

    hints.ai_flags    = AI_PASSIVE;
    hints.ai_family   = family;
    hints.ai_socktype = socktype;

    n = getaddrinfo(hostname, service, &hints, &res);

    if (n <0) {
        fprintf(stderr,
                "getaddrinfo error:: [%s]\n",
                gai_strerror(n));
        return -1;
    }

    ressave=res;

    /*
       Try open socket with each address getaddrinfo returned,
       until getting a valid listening socket.
    */
    sockfd=-1;
    while (res) {
        sockfd = socket(res->ai_family,
                        res->ai_socktype,
                        res->ai_protocol);

        if (!(sockfd < 0)) {
            if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0)
                break;
```

```
            close(sockfd);
            sockfd=-1;
        }
        res = res->ai_next;
    }

    if (sockfd < 0) {
        freeaddrinfo(ressave);
        fprintf(stderr,
                "socket error:: could not open socket\n");
        return -1;
    }

    listen(sockfd, LISTEN_QUEUE);

    freeaddrinfo(ressave);

    return sockfd;
}
```

The TCP daytime server uses `listen_server` with `SOCK_STREAM` and `PF_UNSPEC` parameters to obtain a server socket, which will accept connections to all of the interfaces. When clients connect to the server, it will answer with the daytime information and close the client connection.

File "TCPDayTimeServer.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <string.h>
#include <unistd.h>

#include "listenServer.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int listenfd, connfd;
    socklen_t addrlen;
    char timeStr[256];
    struct sockaddr_storage clientaddr;
    time_t now;
    char clienthost[NI_MAXHOST];
    char clientservice[NI_MAXSERV];

    /* local server socket listening at daytime port=13 */
    listenfd = listen_server( NULL, DAYTIME_PORT,
                              AF_UNSPEC, SOCK_STREAM);
```

```
    if (listenfd < 0) {
         fprintf(stderr,
                 "listen_socket error:: could not create listening "
                 "socket\n");
         return -1;
    }

    for ( ; ;) {
        addrlen = sizeof(clientaddr);

        /* accept daytime client connections */
        connfd = accept(listenfd,
                        (struct sockaddr *)&clientaddr,
                        &addrlen);

        if (connfd < 0)
            continue;

        memset(clienthost, 0, sizeof(clienthost));
        memset(clientservice, 0, sizeof(clientservice));

        getnameinfo((struct sockaddr *)&clientaddr, addrlen,
                    clienthost, sizeof(clienthost),
                    clientservice, sizeof(clientservice),
                    NI_NUMERICHOST);

        printf("Received request from host=[%s] port=[%s]\n",
               clienthost, clientservice);

        /* process daytime request from a client */
        memset(timeStr, 0, sizeof(timeStr));
        time(&now);
        sprintf(timeStr, "%s", ctime(&now));
        write(connfd, timeStr, strlen(timeStr));
        close(connfd);
    }

    return 0;
}
```

The UDP daytime server uses `listen_server` with `SOCK_DGRAM` and `PF_UNSPEC` parameters to obtain a server socket, which will receive connections to all of the interfaces. When clients connect to the server, it will answer with the daytime information and close the client connection.

File "UDPDayTimeServer.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <time.h>
#include <string.h>
```

```
#include "listenServer.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int listenfd, n;
    socklen_t addrlen;
    char *myhost;
    char timeStr[256];
    struct sockaddr_storage clientaddr;
    time_t now;
    char b[256];
    char clienthost[NI_MAXHOST];
    char clientservice[NI_MAXSERV];

    myhost=NULL;
    if (argc > 1)
        myhost=argv[1];

    listenfd= listen_server(myhost,
                            DAYTIME_PORT,
                            AF_UNSPEC,
                            SOCK_DGRAM);

    if (listenfd < 0) {
         fprintf(stderr,
                 "listen_server error:: could not create listening "
                 "socket\n");
         return -1;
    }

    addrlen = sizeof(clientaddr);
    for ( ; ;) {
        n = recvfrom(listenfd,
                     b,
                     sizeof(b),
                     0,
                     (struct sockaddr *)&clientaddr,
                     &addrlen);

        if (n < 0)
            continue;

        memset(clienthost, 0, sizeof(clienthost));
        memset(clientservice, 0, sizeof(clientservice));

        getnameinfo((struct sockaddr *)&clientaddr, addrlen,
                    clienthost, sizeof(clienthost),
                    clientservice, sizeof(clientservice),
                    NI_NUMERICHOST);

        printf("Received request from host=[%s] port=[%s]\n",
               clienthost, clientservice);

        memset(timeStr, 0, sizeof(timeStr));
```

```
        time(&now);
        sprintf(timeStr, "%s", ctime(&now));

        n = sendto(listenfd, timeStr, sizeof(timeStr), 0,
                   (struct sockaddr *)&clientaddr,
                   addrlen);
    }

    return 0;
}
```

The porting process is simple because all main changes are grouped inside `listen_server` function. If application is not correctly structured porting effort increases. Sometimes, it is much better to review the program structure that only to change functions calls to make the adaptation.

This is the reason why scripts to change code automatically are not recommended. Automatic scripts look for concrete functions and change them by new function version however, program structure is not analyzed and many times the result is very poor.

8.2.2 Example: Daytime client (TCP/UDP).

The TCP and UDP versions of the daytime client program use a common function to create the client socket, the `connect_client` function. This function generates a client socket from a hostname, the service, socket family (IPv4 or IPv6) and socket type (TCP or UDP) parameters.

File "ConnectClient.h"

```
#ifndef connect__client__h__
#define connect__client__h__

int
connect_client (const char *hostname,
                const char *service,
                int         family,
                int         socktype);
#endif
```

File "ConnectClient.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "connectClient.h"

int
connect_client (const char *hostname,
                const char *service,
                int         family,
                int         socktype)
{
    struct addrinfo hints, *res, *ressave;
    int n, sockfd;

    memset(&hints, 0, sizeof(struct addrinfo));

    hints.ai_family = family;
    hints.ai_socktype = socktype;

    n = getaddrinfo(hostname, service, &hints, &res);

    if (n <0) {
        fprintf(stderr,
                "getaddrinfo error:: [%s]\n",
                gai_strerror(n));
        return -1;
    }

    ressave = res;

    sockfd=-1;
    while (res) {
        sockfd = socket(res->ai_family,
                        res->ai_socktype,
                        res->ai_protocol);

        if (!(sockfd < 0)) {
            if (connect(sockfd, res->ai_addr, res->ai_addrlen) == 0)
                break;

            close(sockfd);
            sockfd=-1;
        }

        res=res->ai_next;
    }

    freeaddrinfo(ressave);
    return sockfd;
}
```

The TCP daytime server uses `connect_client` with the following input parameter values: `SOCK_STREAM`, `PF_UNSPEC`, the hostname and the port where TCP daytime server is listening. The client socket connects to this server using IPv4 or IPv6 depending on if the server hostname is resolved to an IPv4 or an IPv6 address. Clients will wait for the daytime answer from the server.

File "TCPDayTimeClient.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#include "connectClient.h"

const char *DAYTIME_PORT="13";


int
main(int argc, char *argv[])
{
    int connfd;
    char *myhost;
    char timeStr[256];

    myhost = "localhost";
    if (argc > 1)
        myhost = argv[1];

    connfd= connect_client(myhost,
                           DAYTIME_PORT,
                           AF_UNSPEC,
                           SOCK_STREAM);

    if (connfd < 0) {
         fprintf(stderr,
                 "client error:: could not create connected socket "
                 "socket\n");
         return -1;
    }

    memset(timeStr, 0, sizeof(timeStr));

    while (read(connfd, timeStr, sizeof(timeStr)) > 0)
        printf("%s", timeStr);

    close(connfd);

    return 0;
}
```

The UDP daytime server uses `connect_client` with the following input parameter values: `SOCK_DGRAM`, `PF_UNSPEC`, the hostname and the port where TCP daytime server is listening. And the same as in the TCP case, the client socket connects to this server using IPv4 or IPv6 depending on if the server hostname is resolved to an IPv4 or an IPv6 address. Clients will wait for the daytime answer from the server.

File "UDPDayTimeClient.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>

#include "connect_client.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int connfd, n, m;
    char *myhost;
    char timeStr[256];
    char letter;

    myhost = "localhost";
    if (argc > 1)
        myhost=argv[1];

    connfd = connect_client(myhost,
                            DAYTIME_PORT,
                            AF_UNSPEC,
                            SOCK_DGRAM);

    if (connfd < 0) {
         fprintf(stderr,
                 "client error:: could not create connected socket "
                 "socket\n");
         return -1;
    }

    letter = '1';
    m= write(connfd, &letter, sizeof(letter));

    memset(timeStr, 0, sizeof(timeStr));

    n = read(connfd,
             timeStr,
             sizeof(timeStr));

    printf("%s\n", timeStr);
```

```
    close(connfd);

    return 0;
}
```

8.3 The multicast daytime

In this section daytime utility is used to analyze porting problems related to multicast applications. Also, daytime service is used to show this porting process. Daytime is a very simple service, which is easy to be adapted to multicast.

First some useful functions that will be used in the multicast server/client examples are explained. The `get_addr` function returns a `sockaddr_storage` struct filled with a valid socket address struct for the address, service, family and socket type input parameters. The `joinGroup` function configures the socket with useful multicast options.

The file `mcastutil.h` defines the basic multicast socket interface. It defines the following functions:

- `get_addr` fills the `sockaddress_storage struct`, `addr`, with information related to the hostname, service, family and socktype.

```
int
get_addr (const char *hostname,
          const char *service,
          int         family,
          int         socktype,
          struct sockaddr_storage *addr);
```

- `joinGroup` specifies the address group to be used in the application. It configures the socket with some useful multicast options like `loopBack` and `mcastHop`.

```
int
joinGroup(int sockfd, int loopBack, int mcastHop,
          struct sockaddr_storage *addr);
```

- `isMulticast` checks if an address is a valid multicast group.

```
int
isMulticast(struct sockaddr_storage *addr);
```

File "mcastutil.cpp"

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#include "mcastutil.h"

int
get_addr (const char *hostname,
          const char *service,
```

```
          int         family,
          int         socktype,
          struct sockaddr_storage *addr)
{
    struct addrinfo hints, *res, *ressave;
    int n, sockfd, retval;

    retval = -1;

    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = family;
    hints.ai_socktype = socktype;

    n = getaddrinfo(hostname, service, &hints, &res);

    if (n <0) {
        fprintf(stderr,
                "getaddrinfo error:: [%s]\n",
                gai_strerror(n));
        return retval;
    }

    ressave = res;

    sockfd=-1;
    while (res) {
        sockfd = socket(res->ai_family,
                        res->ai_socktype,
                        res->ai_protocol);

        if (!(sockfd < 0)) {
            if (bind(sockfd, res->ai_addr, res->ai_addrlen) == 0) {
                close(sockfd);
                memcpy(addr, res->ai_addr, sizeof(*addr);
                retval=0;
                break;
            }

            close(sockfd);
            sockfd=-1;
        }
        res=res->ai_next;
    }

    freeaddrinfo(ressave);

    return retval;
}

int
joinGroup(int sockfd, int loopBack, int mcastTTL,
          struct sockaddr_storage *addr)
{
    int r1, r2, r3, retval;

    retval=-1;

    switch (addr->ss_family) {
```

```
    case AF_INET: {
        struct ip_mreq      mreq;

        mreq.imr_multiaddr.s_addr=
            ((struct sockaddr_in *)addr)->sin_addr.s_addr;
        mreq.imr_interface.s_addr= INADDR_ANY;

        r1= setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_LOOP,
                       &loopBack, sizeof(loopBack));
        if (r1<0)
            perror("joinGroup:: IP_MULTICAST_LOOP:: ");

        r2= setsockopt(sockfd, IPPROTO_IP, IP_MULTICAST_TTL,
                       &mcastTTL, sizeof(mcastTTL));
        if (r2<0)
           perror("joinGroup:: IP_MULTICAST_TTL:: ");

        r3= setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                       (const void *)&mreq, sizeof(mreq));
        if (r3<0)
            perror("joinGroup:: IP_ADD_MEMBERSHIP:: ");

    } break;

    case AF_INET6: {
       struct ipv6_mreq    mreq6;

       memcpy(&mreq6.ipv6mr_multiaddr,
              &(((struct sockaddr_in6 *)addr)->sin6_addr),
              sizeof(struct in6_addr));

       mreq6.ipv6mr_interface= 0; // cualquier interfaz

       r1= setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_LOOP,
                      &loopBack, sizeof(loopBack));
       if (r1<0)
           perror("joinGroup:: IPV6_MULTICAST_LOOP:: ");

       r2= setsockopt(sockfd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
                      &mcastTTL, sizeof(mcastTTL));
       if (r2<0)
           perror("joinGroup:: IPV6_MULTICAST_HOPS::  ");

       r3= setsockopt(sockfd, IPPROTO_IPV6,
                      IPV6_ADD_MEMBERSHIP, &mreq6, sizeof(mreq6));
       if (r3<0)
          perror("joinGroup:: IPV6_ADD_MEMBERSHIP:: ");

    } break;

    default:
        r1=r2=r3=-1;
}

if ((r1>=0) && (r2>=0) && (r3>=0))
    retval=0;

return retval;
```

```
}


int
isMulticast(struct sockaddr_storage *addr)
{
    int retVal;

    retVal=-1;

    switch (addr->ss_family) {
        case AF_INET: {
            struct sockaddr_in *addr4=(struct sockaddr_in *)addr;
            retVal = IN_MULTICAST(ntohl(addr4->sin_addr.s_addr));
        } break;

        case AF_INET6: {
            struct sockaddr_in6 *addr6=(struct sockaddr_in6 *)addr;
            retVal = IN6_IS_ADDR_MULTICAST(&addr6->sin6_addr);
        } break;

        default:
           ;
    }

    return retVal;
}
```

File "mcastserver.cpp"

The multicast server is similar to unicast server. It initializes the service, it stops the process (`recvfrom`) waiting for client connections and it answers immediately after any request.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>
#include <time.h>

#include "mcastutil.h"

const char *DAYTIME_PORT="13";

int
main(int argc, char *argv[])
{
    int sockfd, n;
    char *mcastaddr;
    char timeStr[256];
    char b[256];
    struct sockaddr_storage clientaddr, addr;
    socklen_t addrlen;
    time_t now;
```

```
    char clienthost[NI_MAXHOST];
    char clientservice[NI_MAXSERV];

    mcastaddr = "FF01::1111";
    if (argc ==2)
        mcastaddr=argv[1];

    memset(&addr, 0, sizeof(addr));

    if (get_addr(mcastaddr, DAYTIME_PORT, PF_UNSPEC,
                 SOCK_DGRAM, &addr) <0)
    {
        fprintf(stderr, "get_addr error:: could not find multicast "
                "address=[%s] port=[%s]\n", mcastaddr, DAYTIME_PORT);
        return -1;
    }

    if (isMulticast(&addr)<0) {
        fprintf(stderr,
                "This address does not seem a multicast"
                "address [%s]\n",
                mcastaddr);
        return -1;
    }

    sockfd = socket(addr.ss_family, SOCK_DGRAM, 0);

    if (bind(sockfd, (struct sockaddr *)&addr, sizeof(addr)) < 0) {
        perror("bind error:: ");
        close(sockfd);
        return -1;
    }

    if (joinGroup(sockfd, 0 , 8, &addr) <0) {
        close(sockfd);
        return -1;
    }

    addrlen=sizeof(clientaddr);
    for ( ; ;) {
        n = recvfrom(sockfd,
                     b,
                     sizeof(b),
                     0,
                     (struct sockaddr *)&clientaddr,
                     &addrlen);

        if (n <0)
            continue;

        memset(clienthost, 0, sizeof(clienthost));
        memset(clientservice, 0, sizeof(clientservice));

        getnameinfo((struct sockaddr *)&clientaddr, addrlen,
                    clienthost, sizeof(clienthost),
                    clientservice, sizeof(clientservice),
                    NI_NUMERICHOST);
```

```
        printf("Received request from host=[%s] port=[%s]\n",
               clienthost, clientservice);

        memset(timeStr, 0, sizeof(timeStr));
        time(&now);
        sprintf(timeStr, "%s", ctime(&now));

        n = sendto(sockfd, timeStr, sizeof(timeStr), 0,
                   (struct sockaddr *)&addr,
                   sizeof(addr));
        if (n<1)
            perror("sendto error:: \n");

    }
    return 0;
}
```

File "mcastclient.cpp"

The multicast client joins the multicast group and waits announcements received by the multicast group.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <stdio.h>
#include <errno.h>
#include <unistd.h>
#include <stdlib.h>

#include "mcastutil.h"

const char *DAYTIME_PORT="13111";

int
main(int argc, char *argv[])
{
    int sockfd, n;
    char *mcastaddr;
    char timeStr[256];
    char letter;
    struct sockaddr_storage addr, clientaddr;
    int addrlen;
    socklen_t clientaddrlen;


    mcastaddr = "FF01::1111";
    if (argc == 2)
        mcastaddr=argv[1];

    addrlen=sizeof(addr);
    memset(&addr, 0, addrlen);

    if (get_addr(mcastaddr, DAYTIME_PORT,
```

```
                PF_UNSPEC, SOCK_DGRAM, &addr)<0)
    {
        fprintf(stderr, "get_addr error:: could not find multicast "
               "address=[%s] port=[%s]\n", mcastaddr, DAYTIME_PORT);
        return -1;

    }

    sockfd = socket(addr.ss_family, SOCK_DGRAM, 0);

    if (bind(sockfd, (struct sockaddr *)&addr, addrlen) <0) {
        perror("bind error:: \n");
        close(sockfd);
        return -1;
    }

    if (joinGroup(sockfd, 0 , 8, &addr) <0) {
        close(sockfd);
        return -1;
    }

    letter = '1';
    n = sendto(sockfd, &letter, sizeof(letter), 0,
              (struct sockaddr *)&addr,
              addrlen);

    if (n<0) {
        perror("sendto error:: ");
        close(sockfd);
        return -1;
    }

    memset(timeStr, 0, sizeof(timeStr));
    clientaddrlen=sizeof(clientaddr);

    n = recvfrom(sockfd,
                 timeStr,
                 sizeof(timeStr),
                 0,
                 (struct sockaddr *)&clientaddr,
                 &clientaddrlen);


    if (n<0) {
        perror("sendto error:: ");
        close(sockfd);
        return -1;
    }

    printf("%s\n", timeStr);

    close(sockfd);
    return 0;
}
```