Project Number:	**IST-1999-20393**
Project Title:	LONG *Laboratories Over Next Generation Networks*
Deliverable Type:	**P – public**

CEC Deliverable Number:	IST-1999-20393/UPM/WP3.1/DS/P/1/00
Contractual Date of Delivery to the CEC:	M06 (31-May-2001)
Actual Date of Delivery to the CEC:	31-May-2001
Title of Deliverable:	Requirements and guidelines for distributed laboratories application migration
Workpackage contributing to the Deliverable:	WP 3
Nature of the Deliverable:	R – Report
Author(s):	Eva Castro (UPM), Joaquín Salvachua (UPM), Alberto García (UC3M), Carlos Ralli (TID), Ruth Vazquez (TID), Jacinto Vieira (UEV)
Editor:	Tomás P. de Miguel (UPM)

Abstract:	This document provides guidelines for migrating applications and services inside distributed laboratories. The porting section includes the evaluation of strategies, the set up of dual-stack communication framework and the adoption of new communication libraries. The document provides evaluation mechanisms and review of some concrete application porting examples.
Keyword List:	LONG, IPv6, Next Generation Networks, Advanced Network Services, Sockets API

Executive Summary

One important step in the transition from IPv4 to IPv6 is the migration of applications. The transition does not require strong coordination with network migration. Only an small experimental environment should be required to test applications over the new environment. Only during the service deployment phase should be necessary to coordinate activities with network administrators to minimize the number of intermediate dependencies.

This document aims to provide requirements and guidelines for migrating applications and services inside distributed laboratories.

Different sites have different constraints. Sometimes the transition is demanded because of the lack of network addresses, and sometimes the site requires the use of new features provided by IPv6. The IPv6 specification requires 100% compatibility for the existing protocols and applications during the transition.

TABLE OF CONTENTS

1. Introduction

This document was prepared by all partners involved in WP3 (WorkPackage 3) of IST project LONG. It describes guidelines for porting and migrating applications and services into IPv6. This will allow the developers to move smoothly their applications into the new environment.

Different sites have different constraints. Sometimes the transition is demanded because of the lack of network addresses, and sometimes the site requires the use of new features provided by IPv6, such as:

- Support efficient Quality of Service specifications.
- Support of mobile nodes able to move through the network without loosing connectivity.
- IP security level support with authentication and encryption.
- IP multicast support.
- Improve support for autoconfiguration of network addresses.

The transition should be gradual and flexible. The mechanisms for transition [RFC1933] have been designed so that there were as few dependencies as possible between the different elements involved in the transition. All approaches are possible: start with network transition and follow with applications, or vice versa. The main constraint is related to network names. A DNS (and sometimes a NIS server) should be deployed to support both IPv4 and IPv6 addresses before network and application transition is performed.

The most fundamental technique, from the application point of view, used for the transition is known as dual stack. This means that IPv4 nodes are upgraded to support both IPv4 and IPv6. This allows seamless operation with existing applications (that only know about IPv4) as well as perfect interoperability with IPv4 nodes across the network while allowing new applications to take advantage of IPv6.

More information about dual stack machines, other network layer transition mechanisms and scenarios of transition can be found in deliverable 2.1 of LONG project.

Most existing network devices are likely to remain dual for a very long time (many years). However, for some new classes of network devices it might be beneficial to not have to allocate IPv4 addresses, due to the scarcity of globally unique IPv4 addresses. Thus, such devices might rather be soon transformed into IPv6-only ones, given that, while they might have an IPv4 protocol stack, they would not have an IPv4 address assigned to them. Since there is a fundamental constraint in the transition, a node without an IPv4 address can not interoperate at the IP level with IPv4-only nodes. Thus these new network devices would only be able to interoperate with IPv6 capable nodes.

1.1 Scope of the Document

This document is focused on the development of networking applications using any programming language. It includes the definition and description of distributed application scenarios and guidelines to migrate existing applications to new IPv6 environment. The document includes the list of RFCs specifying IPv6 sockets programming interfaces and the comparison with previous ones. This guide also provides examples of code changes.

The document intends to help in the analysis and estimates of the porting, giving guidelines when planning to adopt IPv6. Therefore, it is mainly devoted to distributed program developers, but not only. Network managers can take benefit from the document learning to evaluate their operating service platform migration.

1.2 Relationship with the Overall Project Objectives

One main objective of the LONG project is to evaluate the application migration effort from IPv4 to the new generation IPv6 scenario. WP3 copes with the study, definition and implementation of the functionality and requirements for distributed multimedia applications running over Next Generation networks.

During transition to New Generation networks, the lack of IPv6 applications is one of the factors behind its limited deployment up to now. Therefore, the elaboration of application porting guidelines is essential to accelerate the adoption of IPv6 in the European framework.

The infrastructure deployed in WP2 will serve as a reliable platform to evaluate the characteristics and performance of new applications. The experience gained with the porting and validation of relevant Next Generation applications will be used to produce recommendations on methods to apply when porting existing applications to IPv6 and heterogeneous access scenarios. The guidelines will be validated using several applications both within the project or migrated by others.

This document is devoted to provide guidelines for migrating applications and services, including programming interfaces description for most popular languages.

1.3 Technical Base

There are general recommendations in the applications migration to IPv6. These documents guide developers on the migration of application focusing on the porting of sockets API interface.

The most important one is draft-ietf-ipngwg-rfc2553bis03 (which obsoletes [RFC2553]) which describes the basic socket interface for IPv6. When migration includes the programming of new functionalities related with new available functionalities provided by IPv6, an extended API interface should be used. The advanced functionality is described in [draft-ietf-ipngwg-rfc2292bis-02.txt] (which obsoletes [RFC2292]).

Furthermore, there are another specifications related, depending on application characteristics:

- [RFC1881], *IPv6 Address Allocation Management*. IAB, IESG. December 1995. (Format: TXT=3215 bytes) (Status: INFORMATIONAL).
- [RFC2375], *IPv6 Multicast Address Assignmen*ts. R. Hinden, S. Deering. July 1998. (Format: TXT=14356 bytes) (Status: INFORMATIONAL).

- [RFC2732], *Format for Literal IPv6 Addresses in URL's*. R. Hinden, B., Carpenter, L. Masinter. December 1999. (Format: TXT=7984 bytes) (Status: PROPOSED STANDARD).

- [RFC2894], *Router Renumbering for IPv6*. M. Crawford. August 2000. (Format: TXT=69135 bytes) (Status: PROPOSED STANDARD).

- *Preferred Format for Literal IPv6 Addresses in URL's*, [URL's draft-ietf-ipngwg-url-literal-02.txt] (July 14, 1999).

- "*A SOCKS-based IPv6/IPv4 Gateway Mechanism*", 04/06/2000, [draft-ietf-ngtrans-socks-gateway-04.txt].

- "*Overview of Transition Techniques for IPv6-only to Talk to IPv4-only Communication*", 03/09/2000, [draft-ietf-ngtrans-translator-03.txt].

- "*Basic Socket Interface Extensions for IPv6*", 05/11/2000, <draft-ietf-ipngwg-rfc2553bis-00.txt>.

- "*Advanced Sockets API for IPv6*", 10/28/1999, [draft-ietf-ipngwg-rfc2292bis-02.txt] .

However, all these documents do not provide enough information to allow a smooth migration to the new environment. Not much information is available to give practical experience on how to modify networking applications code to migrate to IPv6. This document aims to provide such practical guide.

2. Scenarios and strategies for global migration

Many information has been written about network migration techniques to the next generation environment. Although there is not too much information about how existing applications should be changed to support IPv6. This section aims to provide information on how to classify applications to estimate the migration effort and provide general guidelines on how to schedule application porting.

Applications should be classified in the following categories, listed in increasing order of complexity to be ported to IPv6:

- **Non networking applications:** Applications that do not establish communication channels with other applications or processes.

- **Site-local networking applications**: Applications that establish communication channels with other applications or processes in the same node.

- **Global networking applications:** Applications that establish communication channels with other applications in other different nodes using IPv4 protocol.

Non networking applications should not be changed when porting to IPv6 networks as they do not need to communicate with any other entity. But this kind of applications could store information related to network communication, for instance, IPv4 addresses. Hence, developers should examine the source code of these applications to check if the allocated memory is sufficient to store the new IPv6 network information. Another typical problem is derived from the different textual format for addresses. Parsers of the text format could fail when porting the application to IPv6.

Site-local networking applications could establish communication channels with other entities, but only in the same node. They should take into account the same issues as non networking applications and, besides, additional ones because of the local communication channels they should create. Many applications could use the loop-back address to establish these local channels. Developers should change the IPv4 loop-back address to the new one. There are other modifications depending on the type of node which runs the site-local networking applications. If the node does not support IPv6, it could only run IPv4 site-local networking applications, so these applications do not need any change. If it is a dual stack node, as a transition mechanism to an IPv6 node, site-local networking applications could transfer information with other IPv6 applications and processes in the same node. As dual stack node provides an environment to let IPv4 and IPv6 interoperation, site-local networking applications do not need any changes either. And finally, if it is an IPv6 node, the source code of these applications should be changed to use, instead of the IPv4 communications API , the API extensions to IPv6.

The main difference between site-local and global networking applications is the communication peer, site-local networking applications use a local peer and global networking applications use a local or remote peer. So, when considering the porting of this kind of applications, the main difference is the resolution of the peer address and the conversion address functions. Source code of global networking applications should be examined to actualize the same changes as site-local applications and the address resolution and conversion functions of the API extensions to IPv6.

The most important requirement in IPv6 migration is that existing applications should continue to work during the migration process. Therefore, the most fundamental technique used for the transition is known as dual stack. This means that IPv4 nodes are upgraded to support both IPv4 and IPv6. Existing applications can interoperate with IPv4 nodes while new application versions can operate with new IPv6 nodes.

This architecture must be maintained during a very long time (many years), because it is not possible to change all applications at the same time. With dual stack it is possible to use both IP address types during the porting period. Developers will only need to port their IPv4 client application to the new IPv6 API and the client application will be able to communicate with both IPv4 only server applications as well as IPv6 server applications running on either a dual host or a v6 only host.

If it is necessary to add new network nodes and there are not enough IPv4 addresses, new IPv6 addresses can be assigned. Dual stack nodes interoperation allow network evolution. Since there is a fundamental constraint in the transition, a node without an IPv4 address can not interoperate at the IP level with IPv4-only nodes. Thus these new network devices would only be able to interoperate with IPv6 capable nodes.

The mechanism to select the appropriate IP version 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.

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 migration period, because not all IPv6-only implementations allow the interaction with any kind of network node, as it can be showed in Table 2-1. Table combinations signed with “X” denote that communication between such kind 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 2-1. 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 can not 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 can not operate over IPv4.

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.

During the gradual transition phase from IPv4 to IPv6, the same application should be run in a IPv4 or IPv6 nodes. 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 library which hides protocol dependency and lets the application source code to be simpler, see Figure 2-1.

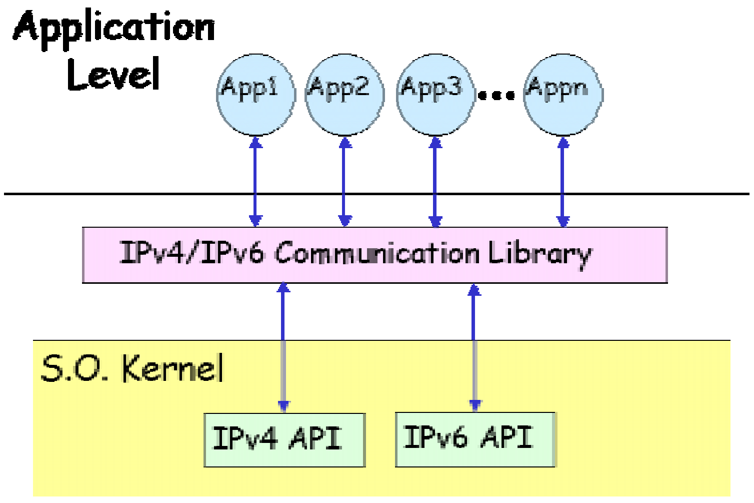


Figure 2-1. Applications protocol independent of IPv4 or IPv6

A communication library allows applications to be independent of the lower level protocol, and provides a common communication interface to every application. Hence, applications can forget about communication problems, since the library charges on behalf of them.

2.1 Other porting considerations

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, following the definitions in [RFC2732] which specifies square brackets to delimit IPv6 addresses: http://[IPv6Address]/index.html

2.2 Porting application modules

The following classes are used in this document to distinguish between different categories of applications or modules inside applications:

- Protocol independent code. The application does not handle IPv6 addresses, i.e. it can not communicate with nodes that do not manage any IP address.
- Networking code is ported by introducing new APIs to replace the old ones.

- Non networking code that needs modifications for some system calls.
- Code that can only be ported if there is a modification of the program logic.

In most distributed applications, the big code deals mainly with logic and algorithmic processing without calling any particular system calls or API related to IP communication. The protocol independent code need not undergo any major modification while porting into IPv6 unless some strategy of processing has changed. These changes could be due to either cater for new demand on new application or to improve the performance of such application in IPv6. Under most normal circumstances, it is recommended that the programmer just do the usual *cut and paste* practice to port the code over to the new application. This gives two major advantages that are: to save time and also, to guarantee the highest degree of compatibility since the original code has been used and debug for some time.

The basic networking code can be ported by just rigidly substituting some of the API and data structures that IPv4 application use to establish and carry out the communication. Such kind of porting is most interesting in its rigid nature of API because its probability of automatically ported is higher than the rest. Chapter 3 is devoted to describe how to proceed to make API substitution.

Besides the system calls and data structures, macros are another issue. For example, to port to IPv6, the macro AF_INET which identify the IPv4 address family can be changed to either AF_INET6 which is the IPv6 specific address family or AF_UNSPEC which is the unspecified address family. Changing to AF_INET6 usually make the porting easier but AF_UNSPEC provides the flexibility of being able to handle multiple address family including both IPv4, IPv6 and others.

Finally, there are certain portion of code that not only affect modification of function calls but also the logic behind the their usage. Such code is the hardest to deal with and cannot be automatically ported most of the time. These codes can be misleading if the programmer does not consider the logic of the program during the migration process.

2.3 IPv4 to IPv6 application reengineering

Up to now, only the idea of application porting has been considered. This implies to substitute IPv4 BSD sockets API calls for these new in the IPv6 socket API. This substitution preserves the IPv4 API semantic and use, so the IPv6 API is a result of a projection from the IPv4 semantic of the API.

One approach to do this consist in apply software reengineering techniques. Application reengineering is usually defined as "*the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form*".

So to change the program implies that the developer understands the logic and structure of the program before starting. This approach can be hard, specially if he is not the author of the code. Of course, it implies that he has access to the source program code.

To perform code reengineering, the developer may start doing some code revision, transforming it from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original, but may be easier to understand for the modification team.

In the porting of the IPv4 applications there can be different degrees. If the application only uses basic communications interface facilities, the identification and migration is easy. However, if advanced facilities should be considered, a redesign of some parts must be done. A taxonomy of porting can be defined depending on the revision degree that should be done:

- Simple API projection: direct substitution of communications API calls.
- Simple program porting: substitution includes revision of small communication code parts.
- Program refactoring: redesign program to encapsulate and isolate the communication code.
- Program reengineering: redesign whole program to add or fix the new functionality.
- Program paradigm change: redesign under a new paradigm or programming language, e.g. from C to C++.
- Program rewritten: starting with the previous code rewrite the code from scratch.
- Program redesign: start from scratch using only the functional requirements.

Note that in some cases the last solutions will be better than to keep the old code.

One of the current methodologies for reengineering and refactoring is the Extreme Programming. It is heavily based on Object Oriented programming, but because of its principles it can be applied to this case.

SUN Microsystems has developed the "Socket Scrubber" tools to make easier the upgrade from IPv4 to IPv6. These tools search for predefined lists of IPv4 keyword parameters and function calls in the source code and return the code section where changes should be made. The result of Socket Scrubber execution depends on source code, which should use the coding standards instead of hard-coding contents for parameters. Only a pattern matching for the keyword strings is checked.

The following references contain more information on this issue:

- http://satc.gsfc.nasa.gov/support/index.html
- http://www.sei.cmu.edu/reengineering/
- http://www.comp.lancs.ac.uk/projects/RenaissanceWeb/Reengineering/
- http://www.tcse.org/revengr/
- http://www.sei.cmu.edu/reengineering/pubs/iwsa95/

2.4 Solutions without sources

The previous solutions only work if the source codes are available. In some environments, the reverse engineering of the application code can even be illegal, so a way for bypassing this situation must be found.

Sometimes Reverse Engineering technique can be applied. This will heavily depend on the license owned for the software. Usually, being an end user for the software most of software licenses advice not to do reverse engineering of the code. Only in some cases this will be a feasible way to go.

Reverse Engineering tools allow the substitution of IPv4 API library calls by equivalent IPv6 calls. This procedure has a lot of restrictions because it is not possible to analyze high level design structure without source code.

If the application uses standard services, the approach of applications and services migration guidelines is to build the IPv6-compliant server:

The suite of building IPv6-compliant server includes building Web server, FTP server, MAIL server, DNS server, etc.

3. Communication APIs

This section is aimed to discuss about the main issues that application developers should take into account when porting networking code to IPv6. Using new IPv6 features usually requires a new application architecture design which is out of the scope of this section.

Application porting must be written into certain programming language. Usually, the programming languages have some libraries or API to access the network facilities and to use them to implement the communication. This section is focused on the description of key aspects to take into account, and on the development status of new communication libraries for the most popular programming languages.

The IPv6 API can be divided in two main parts: the basic interface to replace the IPv4 API and the advanced features API which provide access to new facilities available only in IPv6.

3.1 Porting of C programs

3.1.1 Berkeley socket interface

Since Berkeley socket interface was created in the early 80s, it has became the de facto standard for network programming. Many Unix systems use this API as the base for their systems, for instance FreeBSD. Other, as Linux, developed their networking code and the socket API from scratch.

Some changes are needed to adapt the socket API for IPv6 support. They are fully described in [RFC2553]. This section gives a briefly description of these changes, more information can be found in Appendix A.

The main differences between socket interface API and the extensions to IPv6 are:

- Data structures.
- Name-to-address functions.
- Address conversion functions.
- Core sockets functions.

3.1.1.1 Data structures

The size of the IP address is visible to applications through the socket address structures in the socket interface. Hence, a new definition of the socket address structure is required, `sockaddr_in6`. The `sockaddr_in` structure is the protocol specific address data structure for IPv4.

The next Figure 3-1 shows the differences between `sockaddr_in`, `sockaddr_in6` and the generic socket address structure, sockaddr. Socket functions are defined as taking a pointer to a generic socket address structure as an argument, since they must deal with socket address structures from any of the supported protocol families. Calls to these functions must cast the pointer to the protocol specific socket address structure to be a pointer to a generic address structure.

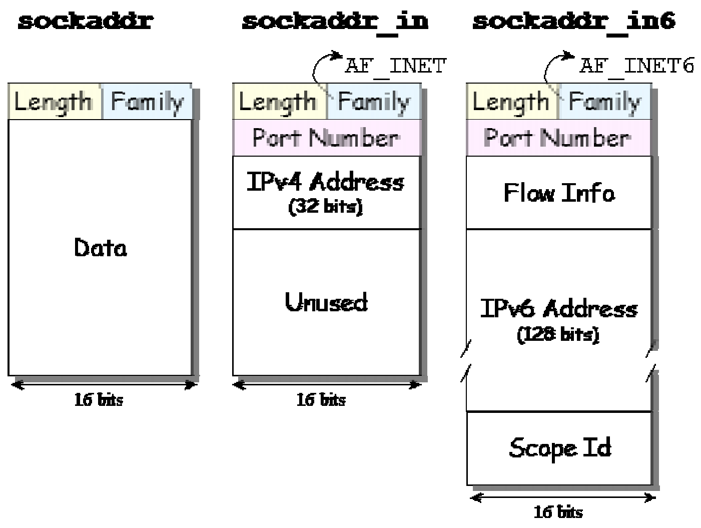


Figure 3-1. Comparison of socket address structures

In order to provide portable code across multiple address families and platforms, developers should define a new structure with adequate length and alignment to support many protocol-specific address structures, `sockaddr_storage` (proposed at draft-ietf-ipngwg-rfc2553bis-00.txt).

3.1.1.2 Name-to-address functions

Applications frequently use names instead of numeric addresses for hosts. There are defined two new thread safe functions to make the translation: `getipnodebyname` and `getipnodebyaddr`. When using the former, we get the IP address from the hostname or the numeric address string (i.e, a dotted-decimal IPv4 address or an IPv6 hex address), and when calling the latter, we get the hostname from the IP address. These functions are protocol dependent, both take as an argument the address family to configure a query for an IPv4 or IPv6 query.

There are defined two new function, `getaddrinfo` and `getnameinfo`, which hide all of the protocol dependencies. Hence, applications should only work with the socket address structures that are filled by these functions. The `getaddrinfo` function returns a set of socket addresses, which match with the name and/or service given, and additional information to be used in creating a socket. The `getnameinfo` returns the hostname and service associated to a socket address structure.

3.1.1.3 Address conversion functions

The new functions `inet_pton` and `inet_ntop` convert from ASCII string into the binary address format, the socket address structure, and vice-versa. The original ones `inet_aton` and `inet_ntoa` can only manage 32-bit IP addresses, hence, they can not be used with IPv6 addresses.

3.1.1.4 Core socket functions

There are not differences between IPv4 and IPv6 socket functions. The only differences are the values which these functions are called.

The socket function creates a descriptor for a network communication. This function takes different values for the address family and the protocol type arguments depending on the IP protocol version. Applications use the rest of socket API functions to establish communication channels. With this purpose, kernel and applications need exchange socket address structures information using sockets functions:

Applications pass an address structure into the system kernel, see Figure 3-2. Applications must cast from a specific protocol address structure to a generic address structure, `sockaddr`. Examples of this kind of API fuctions are: `bind, connect, sendmsg` and `sendto`.

Table 3-1. BSD socket API

Function	Meaning
Accept	Locks the server until a connection request arrives. When called it returns the remote end IP address that has established the connection. Besides, it returns a descriptor (integer) of the connected socket; this number is not the same to the number of the listening socket in the well-known port of the server. This mechanism provides the server the ability to go on receiving connection requests from other clients and so that it could provide concurrency.
Bind	Binds the socket to the local server IP address and the listening port. There are some ports already reserved to a definite set of services, these are known as well-known TCP/UDP port numbers. TCP ports do not conflict with UDP ports because each protocol has its own port space.
Close	Makes the communication channel free. It also makes the resource occupied by the socket available again.
Connect	Establishes a connection to a remote transport end point (IP address and port). From this moment on, the socket can send/receive information to/from this end.
Gethostname	Gets the name of the local machine.
Getpeername	Gets the remote end point address for a connected socket.
Listen	Changes the socket to a passive socket if it is not connected (with connect()) and so, the OS kernel will accept remote connections to this socket. By default, when sockets are created (with socket()), the OS treats them as active sockets.

Recvfrom	Waits for some data from an UDP transport end point. Once it is called, the application is locked waiting for some data to arrive.
Read	Waits for receiving some information coming from a TCP transport end point.
Recvmsg	Receives a message from a remote end point.
Sendmsg	Sends a message to a remote end point.
Sendto	Used by the UDP application to send data to a remote end point.
Socket	Gets a TCP/UDP communication channel. When calling this function the family socket (either AF_INET) and the kind of connection (SOCK_STREAM , SOCK_DGRAM) have to be specified.
Write	Sends the content of a memory buffer through a socket. There is no need to specify the remote socket address because there is a connection already established and the OS already knows the IP address and port where data is destined to.

The kernel returns an address structure to the application, see Figure 3-2. Applications must cast from a generic address structure, sockaddr, to a specific protocol address structure. Examples of this kind of API functions are: accept, recvfrom, recvmsg, getpeername and getsockname.

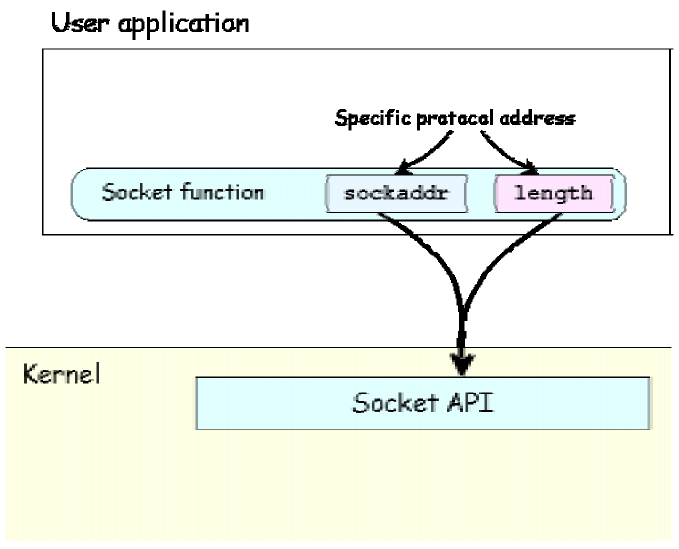


Figure 3-2. Generic address structure from an application to the kernel

Note, in both cases, the information exchanged is a pointer to a generic socket address structure, `sockaddr`. Applications must cast this generic structure to the protocol specific socket address structure using the provided structure length when receiving it from kernel.

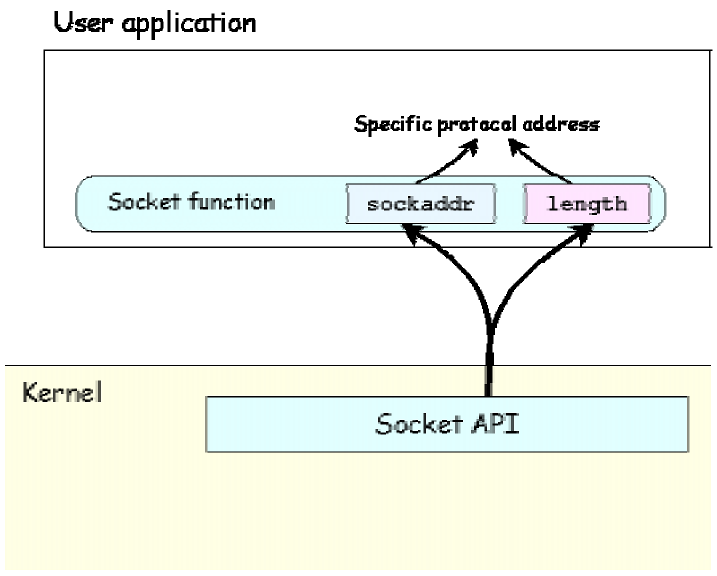


Figure 3-3. Generic address structure from the kernel to an application

Always a host wants to establish an UDP communication channel to a remote host, it uses the same socket functions independently of the IP network protocol used, see Figure 3-4.

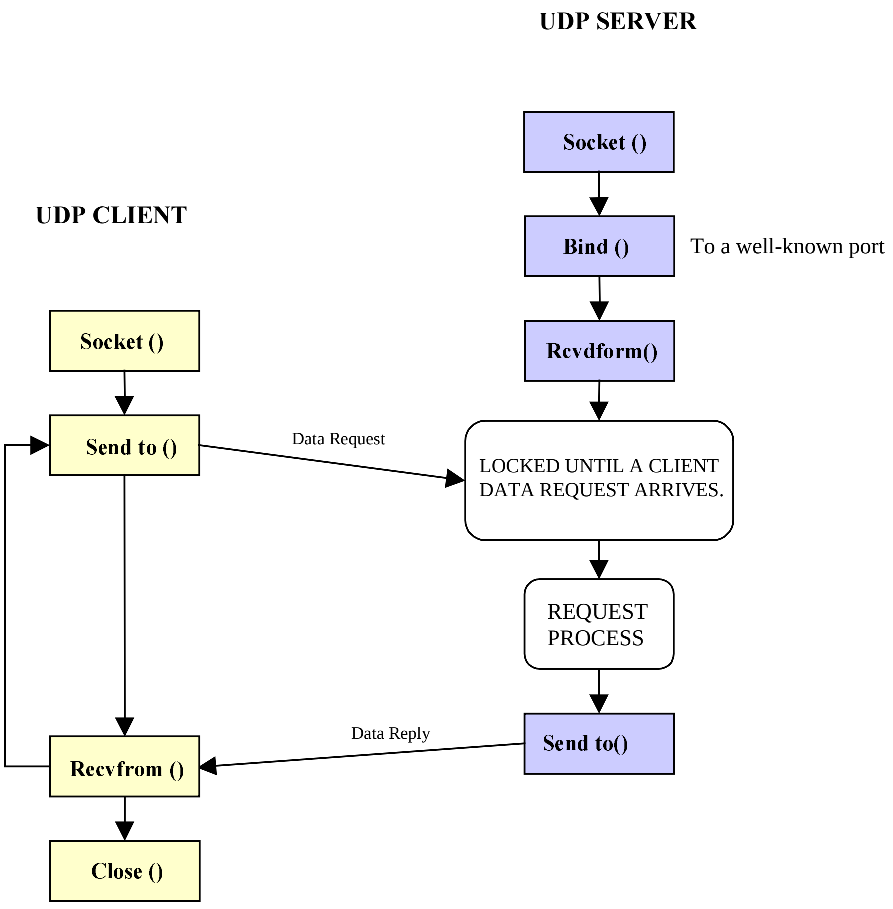


Figure 3-4. UDP Client/Server Programming Model

The Figure 3-5 shows how a TCP connection is developed and makes easy to understand the whole process of programming under the TCP client/server model. The only difference between IPv4 and IPv6 relies in data structures, so this figure is worth the same for IPv4 and IPv6.

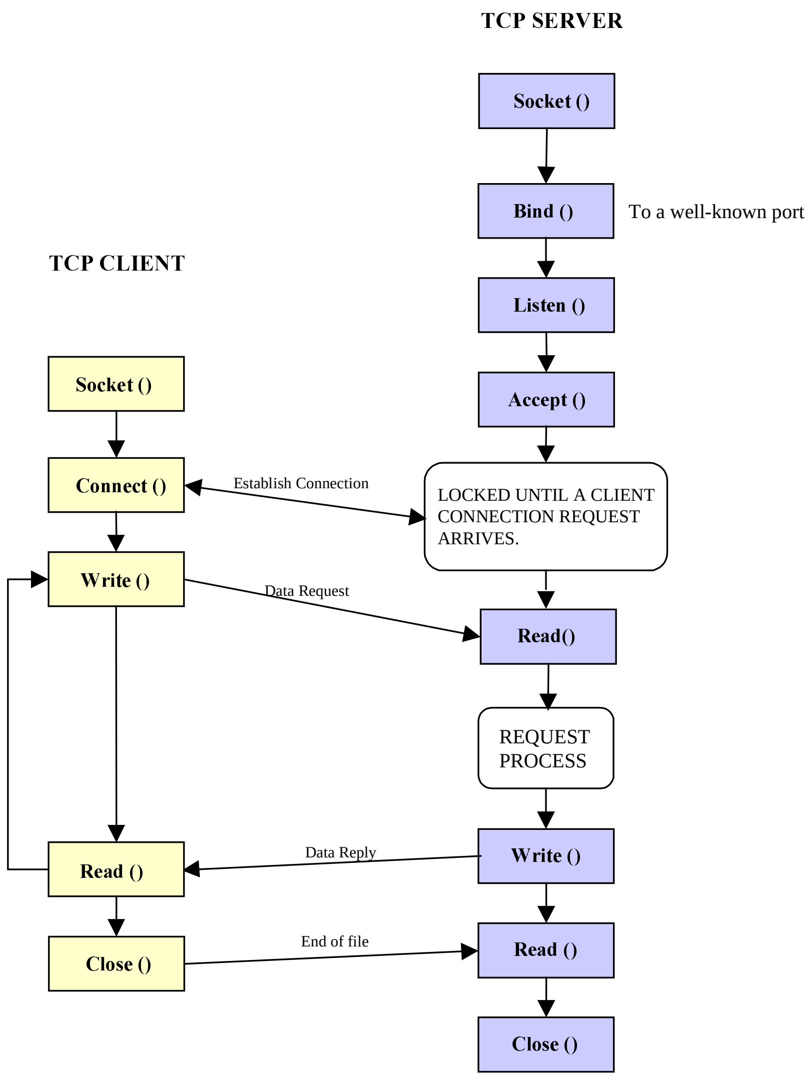


Figure 3-5. TCP Client/Server Programming Model

3.1.2 Winsock interface

In this section, we discuss what Winsock is, and how it can be used by applications. Also, the portability to IPv6 of an IPv4 application, that was written using the Winsock, is explained.

Winsock – Windows Socks – is a network application-programming interface (API) for Microsoft Windows. In WinSock, set of data structures and function are implemented as a Dynamic Link Library (DLL).

Winsock uses the sockets paradigm that was first implemented by Berkeley Software Distribution (BSD) and later, adapted for Microsoft Windows as Windows Sockets 1.1. This version was the first official release but soon became the industry standard.

The WinSock 1.1 specification focuses exclusively on the IPv4 protocol families. There are 16-bit and 32-bit versions of WinSock1.1, and it is included in the of Microsoft Operating System.

A new version was implemented to expand the functionalities of the original standard. This new version is called Windows Sockets Version 2 – WinSock 2 – and supports other protocol suites, such as IPv6, ATM, IPX/SPX and DECnet. WinSock2 was designed to be protocol independent. Besides WinSock 2 provides new functionality, it is backward compatibly with version 1.1.

WinSock 2 allows the coexistence of multiple protocol stacks and the creation of application that are network protocol independent. That is, an application can adapt to various network environments using the mechanisms that WinSock2 provides.

The implementation of applications that support IPv6 is only possible using the WinSocks 2 services.

3.1.2.1 Winsock Architecture

WinSock (version 2) is based in the Open Services Architecture (WOSA) model. This model defines a standard service provider interface (SPI) between the Windows Socket interface, implemented as dynamic link library (WinSock DLL), and protocol stacks.

The API is specified for application developers and the SPI is specified for protocol stack and namespace service providers.

In this way, a single WinSock DLL can simultaneously access multiple stacks provide by different vendors. In opposition, what happens with Winsock 1.1, that it is the transport protocol vendor that supplies the WinSock libraries. Besides this fact, WinSock 2 is backward compatible with WinSock 1.1. The Winsock Architecture is illustrated in Figure 3-6.

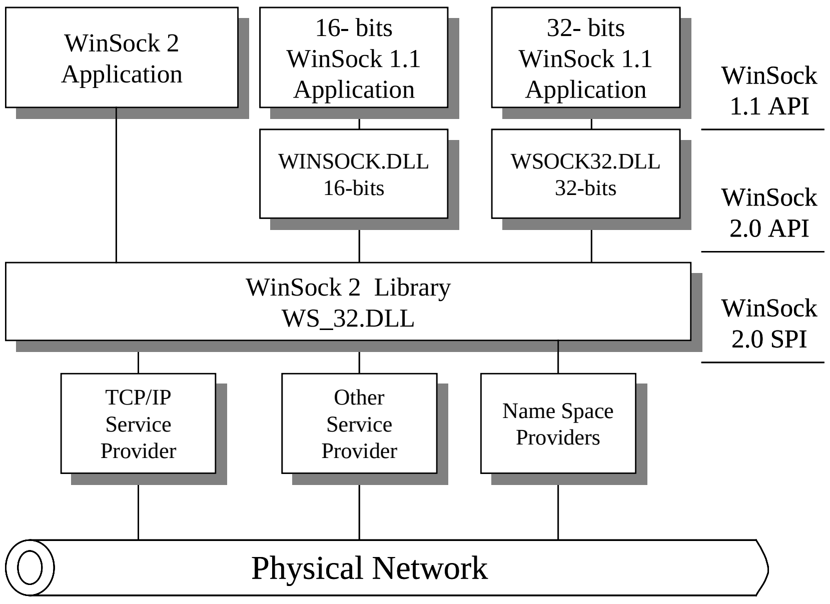


Figure 3-6 - Windows Socks architecture

Also, Winsock 2 includes functions that standardize the access to network naming services. To perform this, a name-space provider is included in the Winsock architecture. This module is responsible of name resolution services. In this way, an application can perform name resolution without having to know the details of how a particular name service works. WinSock can handle multiple name-space providers at the same time.

3.1.2.2 WinSock distributions

There are currently three distributions of Winsock. In Table 3-2 the files associated with each distribution are shown.

All operating systems of Microsoft have included the Winsock. WinSock 1.1 is included on Windows 95 and earlier versions of NT 4. WinSock 2.0 is included on other versions.

The majority C and C++ compilers provide the header files and libraries needed for Winsock to Windows.

Table 3-2. WinSock Files

Platform	Application	Dynamic Link Library	Development Files
16 or 32 bits Windows	16-bit WinSock 1.1	WINSOCK.DLL	WINSOCK.H WINSOCK.LIB
32 – bit Windows	32-bit WinSock 1.1	WSOCK32.DLL	WINSOCK.H WSOCK32.LIB
32 – bit Windows	32-bit WinSock 1.1	WS2_32.DLL	WINSOCK2.H WS2_32.LIB

3.1.2.3 Porting from Winsock 1.1 to WinSock 2

As it says before, WinSock 2 is compatible with version 1.1. This compatibility is done on two levels: source and binary.

Source code compatible means that an application can be re-compiled to run on Winsock 2. For that, only is needed to include the new header file, Winsock2.h and link with the appropriate WinSock 2 libraries (ws_32.lib rather than winsock.lib or wsock32.lib).

Binary compatibility means that Winsock 2 fully supports WinSock 1.1 executables. To guarantee this compatibility, WinSock 2 includes two dynamic link libraries that provide a WinSock 1.1 interface (see architecture in Figure 3-6). Applications that use the Winsock 2 API make call directly into ws_32.dll. For the applications that use Winsock 1.1 API, the winsock.dll and wsock32.dll or wsock32.dll are used.

In both cases, a properly installed TCP/IP service provider is needed.

3.1.2.4 WinSock Extension to the Berkely API

Winsock (version 2) includes the complete Berkely sockets API. Also, it provides a number of extensions to this standard to allow asynchronous access to network events, as well as enable overlapped I/O. These extension functions are prefixed with the letters WSA.

The following tables summarize the functions included in Windows Sockets 2: the Table 3-3 presents Berkeley-style functions; the Table 3-4 presents the Microsoft Windows-specific Extension functions; and the Table 3-5 presents name registration and resolution function.

3.1.2.5 Socket Functions

The Windows Sockets specification includes all the following Berkeley-style socket routines that were part of the Windows Sockets 1.1 API.

Table 3-3. Berkeley-style socket routines

Routine	Meaning
`Accept`[1]	An incoming connection is acknowledged and associated with an immediately created socket. The original socket is returned to the listening state.
`Bind`	Assigns a local name to an unnamed socket.
`Closesocket`	Removes a socket from the per-process object reference table. Only blocks if `SO_LINGER` is set with a nonzero time-out on a blocking socket.
`Connect`[1]	Initiates a connection on the specified socket.
`Getpeername`	Retrieves the name of the peer connected to the specified socket.
`Getsockname`	Retrieves the local address to which the specified socket is bound.
`Getsockopt`	Retrieves options associated with the specified socket.
`Htonl`[2]	Converts a 32-bit quantity from host-byte order to network-byte order.
`Htons`[2]	Converts a 16-bit quantity from host-byte order to network-byte order.
`Inet_addr`[2]	Converts a character string representing a number in the Internet standard "." notation to an Internet address value.
`Inet_ntoa`[2]	Converts an Internet address value to an ASCII string in "." notation that is, "a.b.c.d".
`Ioctlsocket`	Provides control for sockets.
`Listen`	Listens for incoming connections on a specified socket.
`Ntohl`[2]	Converts a 32-bit quantity from network-byte order to host-byte order.
`Ntohs`[2]	Converts a 16-bit quantity from network byte order to host byte order.
`Recv1`	Receives data from a connected or unconnected socket.
`Recvfrom`[1]	Receives data from either a connected or unconnected socket.
`Select1`	Performs synchronous I/O multiplexing.
`Send1`	Sends data to a connected socket.
`Sendto`[1]	Sends data to either a connected or unconnected socket.
`Setsockopt`	Stores options associated with the specified socket.
`Shutdown.`	Shuts down part of a full-duplex connection
`Socket`	Creates an endpoint for communication and returns a socket descriptor.

[1] The routine can block if acting on a blocking socket.

[2] The routine is retained for backward compatibility with Windows Sockets 1.1, and should only be used for sockets created with AF_INET address family.

Microsoft Windows-Specific Extension Funtions

The Windows Sockets specification provides a number of extensions to the standard set of Berkeley Sockets routines.

Table 3-4. Microsoft Windows-specific Extension functions

Routine	Meaning
`WSAAccept`[1]	An extended version of accept, which allows for conditional acceptance.
`WSAAsyncGetHostByAddr` [2,3]	A set of functions that provide asynchronous versions of the standard Berkeley getXbyY functions. For example, the WSAAsyncGetHostByName function provides an asynchronous, message-based implementation of the standard Berkeley gethostbyname function.
`WSAAsyncGetHostByName` [2,3]	
`WSAAsyncGetProtoByName` [2,3]	
`WSAAsyncGetProtoByNumber` [2,3]	
`WSAAsyncGetServByName` [2,3]	
`WSAAsyncGetServByPort` [2,3]	
`WSAAsyncSelect`[3]	Performs asynchronous version of select.
`WSACancelAsyncRequest`[2,3]	Cancels an outstanding instance of a WSAAsyncGetXByY function.
`WSACleanup`	Signs off from the underlying Windows Sockets .dll.
`WSACloseEvent`	Destroys an event object.
`WSAConnect`[1]	An extended version of connect which allows for exchange of connect data and QOS specification.
`WSACreateEvent`	Creates an event object.
`WSADuplicateSocket`	Allows an underlying socket to be shared by creating a virtual socket.
`WSAEnumNetworkEvents`	Discovers occurrences of network events.
`WSAEnumProtocols`	Retrieves information about each available protocol.
`WSAEventSelect`	Associates network events with an event object.
`WSAGetLastError`[3]	Obtains details of last Windows Sockets error.
`WSAGetOverlappedResult`	Gets completion status of overlapped operation.
`WSAGetQOSByName`	Supplies QOS parameters based on a well-known service name.
`WSAHtonl`	Extended version of htonl.
`WSAHtons`	Extended version of htons.
`WSAIoctl1`	Overlapped-capable version of IOCTL.

WSAJoinLeaf[1]	Adds a multipoint leaf to a multipoint session.
WSANtohl	Extended version of ntohl.
WSANtohs	Extended version of ntohs.
WSAProviderConfigChange	Receives notifications of service providers being installed/removed.
WSARecv[1]	An extended version of recv which accommodates scatter/gather I/O, overlapped sockets and provides the flags parameter as in, out.
WSARecvFrom[1]	An extended version of recvfrom which accommodates scatter/gather I/O, overlapped sockets and provides the flags parameter as in, out.
WSAResetEvent	Resets an event object.
WSASend[1]	An extended version of send which accommodates scatter/gather I/O and overlapped sockets.
WSASendTo[1]	An extended version of sendto which accommodates scatter/gather I/O and overlapped sockets.
WSASetEvent	Sets an event object.
WSASetLastError[3]	Sets the error to be returned by a subsequent WSAGetLastError.
WSASocket	An extended version of socket which takes a WSAPROTOCOL_INFO structure as input and allows overlapped sockets to be created.
WSAStartup[3]	Initializes the underlying Windows Sockets .dll.
WSAWaitForMultipleEvents1	Blocks on multiple event objects.

[1] The routine can block if acting on a blocking socket.
[2] The routine is always realized by the name resolution provider associated with the default TCP/IP service provider, if any.

Name Registration and Resolution function

The Windows Sockets specification provides functions for the network naming services.

Table 3-5. Name Registration and Resolution Function

Function	Description
WSAAddressToString	Converts an address structure into a human-readable numeric string.
WSAEnumNameSpaceProviders	Retrieves the list of available Name Registration and Resolution service providers.
WSAGetServiceClassInfo	Retrieves all of the class-specific information pertaining to a service class.
WSAGetServiceClassNameByClassId	Returns the name of the service associated with the given type.
WSAInstallServiceClass	Creates a new new service class type and stores its class-specific information.

`WSALookupServiceBegin`	Initiates a client query to retrieve name information as constrained by a WSAQUERYSET data structure.
`WSALookupServiceEnd`	Finishes a client query started by WSALookupServiceBegin and frees resources associated with the query.
`WSALookupServiceNext`	Retrieves the next unit of name information from a client query initiated by WSALookupServiceBegin.
`WSARemoveServiceClass`	Permanently removes a service class type.
`WSASetService`	Registers or removes from the registry a service instance within one or more name spaces.
`WSAStringToAddress`	Converts a human-readable numeric string to a socket address structure suitable for passing to Windows Sockets routines.

3.1.2.6 Porting Winsock compliant applications to IPv6

The changes performed on the sockets functions so that they support IPv6 are documented [RFC 2553]. In the same way, the Winsock (only version 2) supports [RFC 2553] with some exceptions:

- The header files listed in this RFC are not applicable. Instead of these, winsock2.h, ws2tcpip and tcpipv6.h header files should be used.
- Winsock enables only the Socket Address Structure compatible with 4.3BSD-Based System and defined in section 3.3 in the [RFC2553].

  The structure compatible with 4.4 BSD Based System, and defined in 3.4, does not apply.
- IPv4-mapped addresses as described in section 3.7 are not supported. According to this section the IPv4-mapped addresses can be generated automatically by the getipnodebyname() function when the specified host has only IPv4 address.
- The interface identification functions described in section 4 are not supported. The identification of the interfaces as described in this section is only applied on Unix-based systems.
- The new functions to perform operations with IPv6 address, as defined in section 6.1-6.3 and 6.6, are not supported.

In sections 6.1-6.3, new functions were defined to replace `gethostname()` and `gethostbyaddr()`, such as `getipnodebyname()`, `getipnodebyaddr()` and `freehostent`. These functions are not supported.

In same way, the new functions defined to replace `inet_addr()` and `inet_ntoa()` as defined in section 6.6 are not supported. In this sections are proposed `inet_nton()` and `inet_ntop()`.

Microsoft makes available a tool called Checkv4.exe[i] to scan source code files and identify code that needs to be changed to support IPv6.

3.2 Porting of C++

C++ is an object oriented programming language which is the mostly upwardly compatible extension of C. Porting C++ applications to IPv6 is similar to C applications case and the network programming API depends on the operating system which applications work. The main advantage of working with C++ is the set of properties derived from using an object oriented programming language: overloading, polymorphism, template definition and dynamic binding. This section is aimed to show how developers could use these features to porting C++ applications using the socket API, although these concepts may be applied to the rest of APIs.

Most applications use TCP and UDP transport protocols instead of using the IP network layer protocol directly. From application point of view, applications do not need to know which version of IP protocol are using. So, application code should be protocol independent. Unfortunately, as we have seen in the last section, there are protocol dependent issues which are visible to applications through socket API. Inheritance mechanism of the object oriented programming languages provides a way to define a set of classes containing the general properties of TCP and UDP services, hiding the specific details of the network layer protocol. Typically, this set of related classes can be grouped into a class library, a “network standard component”. One of the advantages of this network component definition is reusability. Other network applications may share this library to use TCP and UDP services. Although reusability is not dependent on specific language features, C++ features make reusability easier.

3.2.1 Case of study

This case of study is aimed to show basic IPv6 features required to change existing TCP and UDP network applications. Most network application only use these basic features, so changes are based on: the larger address size and the basic socket functions.

Figure 3-7 shows the UML diagram of a network class library example. There is a wrapper for each one of the TCP and UDP transport protocols: `streamSocket_t` and `dgramSocket_t`. These wrappers are used to establish communication channels between network endpoint addresses, `inetAddr_t`. Note, as a generic network class library , it may provide interfaces for other communication protocols, in the Figure 3-7 the Unix domain protocols.

[i] Checkv4.exe is shipped with IPv6 Technology Preview files and it is found in the \bin folder.

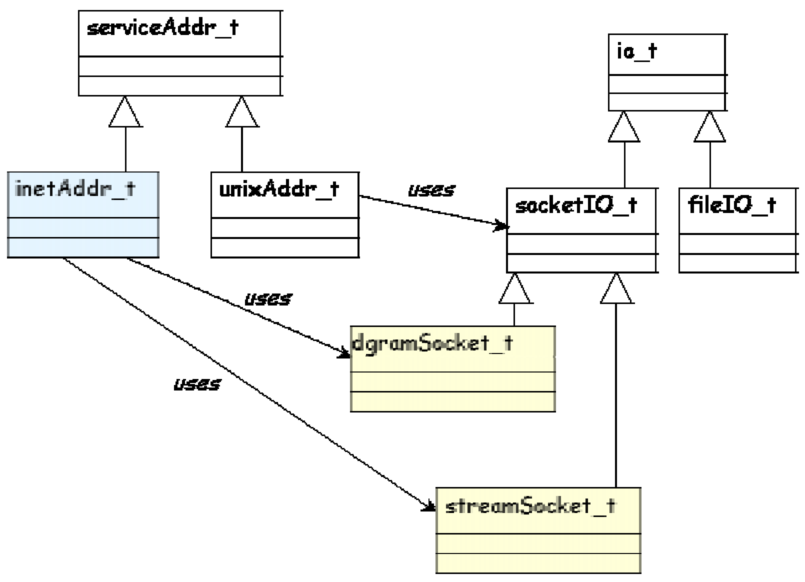


Figure 3-7. UML diagram of a network class library example

Both, `streamSocket_t` and `dgramSocket_t` store the network endpoint addresses as attributes and have methods to wrapper the C API functions providing the same functionality.

When an application need a TCP or UDP channel, it should create first the network endpoint address (or two addresses, depending on using server o client model), typically from a hostname and service port, and then it should create the `streamSocket_t` or `dgramSocket_t` wrapper.

Developers should easily extend this API to support IPv6 protocol only changing the IP address structure. They could use one of the following solutions:

Create a new address class for IPv6 addresses, `inet6Addr_t`, inherited from `serviceAddr_t`.

Change `inetAddr_t` to support IPv6 addresses.

First solution seems clearer than the second one. However, if the first one is used, applications should know which IP protocol version are using in order to create `inet6Addr_t` or `inetAddr_t` structure, they are protocol dependent. Using the second solution, applications create an `inetAddr_t` structure, from a hostname and service port, and the network library is the burden of creating an IPv6 address if this protocol is supported, or an IPv4 one in the other case.

Methods of `streamSocket_t` and `dgramSocket_t` classes which are wrappers of the socket API functions should not be changed, as they are called using the generic socket address structure defined in the socket API.

Once the network library supports IPv6 protocol, old applications using this library work properly on IPv4 and IPv6 systems with minimal changes in application source code.

Portability is guaranteed since library supports both IPv4 and IPv6 protocols. If applications run on a system that does not support IPv6, the IPv6 socket creation will fail, but the library will handle this error creating an IPv4 socket and finally the communication channel will succeed.

3.3 Migration of SOCKS based applications

SOCKS is a popular protocol that is used to control generic traffic through firewalls.

A SOCKS server is located at the firewall. Applications (using any application protocol) on the internal network connect to the SOCKS server and, in turn, the SOCKS server connects to the outside world. The SOCKS server relays communications between the internal client application and the outside world. The SOCKS server can permit/deny any connection based on rules that an administrator defines: what these rules are or how they are specified is not part of the SOCKS protocol. The applications outside the firewall think they are talking to a normal client.

SOCKS operates at the connection-level, so that any application protocol can flow through SOCKS. In contrast, higher-level application proxying can only work with their corresponding application protocols – for example, an HTTP proxy server will only relay HTTP protocol traffic. The advantage of SOCKS is that it is generic – the advantage of application proxies is that they have a detailed understanding of the

application protocol and so, they can be more specific in auditing and rules regarding data flows can be tied to the protocol. Many firewalls contain both a SOCKS server for generic traffic and application proxies for the most popular protocols (HTTP).

SOCKS should not be confused with the sockets API. Most software engineers are familiar with the sockets API for network programming.

Other mechanism for porting applications is the use of socks to make the transition. This mechanism is described in [RFC3089] "*A SOCKS-based IPv6/IPv4 Gateway Mechanism*". In the document the use of a socks server as an IPv4 to IPv6 gateway is described. Instead of using socks like a secure way for the applications to cross firewalls and to access to intranets, the same protocol is used for the gateway approach.

Several applications that are already “sockified” will be usable in some IPv6 environment without any modification.

Services of a SOCKS Server

The SOCKS specification describes the protocol between a SOCKS server and a SOCKS client. It does not mandate what range of services the SOCKS server provides, beyond stating that it communicates with the outside world.

As the SOCKS server is a single conduit through which internal-external transmissions flow, it is the ideal spot to monitor, control and audit what is happening. This is the real reason SOCKS is deployed. SOCKS servers are available stand-alone, built into proxy servers and built into firewall products. They offer easy-to-use tools to setup SOCKS connections, define rules that automatically permit/deny connections and configurable recording of information about what flows through the SOCKS server.

A free implementation for a SOCKSv5 proxy server can be found at http://www.inet.no/dante/.

Services of a SOCKS Client

Client applications running on devices inside a firewall will often have to communicate with other devices inside the firewall, and with servers outside the firewall. If the firewall has SOCKS deployed, then clients seeking outside connections need to communicate with the SOCKS server which, in turn, communicates with the remote server. There are two main techniques in use for this:

- The first option is to completely replace the sockets library on the client device with a SOCKS compliant layer that transparently adds SOCKS functionality to the underlying sockets APIs. The advantage here is that applications can use SOCKS without having to be re-coded.
- The second option is to create a new API that applications can selectively call. The advantages of this is that client apps that need to talk to server outside the server can use SOCKS, but other applications on the same machine that only need to talk to other devices inside the firewall need not use SOCKS.

Adding SOCKS

Following the recommendations at http://www.socks.nec.com we can port an application to use socks by compiling with SOCKSv5:

- The library of SOCKS v5 Reference Implementation has functions that are equivalent to the standard BSD socket functions. Converting an application to use the SOCKS protocol normally only requires adding an #include directive and linking with this library. Applications that follow the SOCKS friendly guidelines should work without any further modification.
- Identify the program for syslog.
- Add this line at or near the beginning of the main procedure:

  ```
  SOCKSinit(argv[0]);
  ```

  If you omit this line, the syslog lines that display on the client host describe a generic program name, instead of the actual client program name.
- Map functions to the SOCKSified replacement functions SOCKS V4 users: Add these

#define directives to all cc lines:

```
-Dconnect=Rconnect -Dgetsockname=Rgetsockname \
-Dgetpeername=Rgetpeername -Dbind=Rbind \
-Daccept=Raccept -Dlisten=Rlisten -Dselect=Rselect
```

SOCKS v5 users

- Add these #define directive to all cc lines and include the socks.h header file:

```
-DSOCKS
```

- If you use a makefile, add this line to the CFLAGS macro definition.
- Link with the appropriate SOCKS library.
- Add these to the command that creates the executable:

```
SOCKS V4 users:
-L<socks_lib_dir> -lsocks
SOCKS v5 users:
-L<socks_lib_dir> -lsocks5
```

The SOCKS V4 client library cannot SOCKSify UDP sockets because SOCKS V4 doesn't support UDP.

Converting SOCKS V4 to SOCKS v5

- If the source code is written using the SOCKS V4 library, compile it with the library of SOCKS v5 Reference Implementation.
- Map functions to the replacement functions of SOCKS v5 Reference Implementation.
- To all cc lines, add:

```
-DRconnect=SOCKSconnect \
-DRgetsockname=SOCKSgetsockname \
-DRgetpeername=SOCKSgetpeername \
-DRbind=SOCKSbind \
-DRaccept=SOCKSaccept \
-DRlisten=SOCKSlisten \
-DRselect=SOCKSselect \
-Drecvfrom=SOCKSrecvfrom \
-Dsendto=SOCKSsendto \
-Drecv=SOCKSrecv \
-Dsend=SOCKSsend \
-Dread=SOCKSread \
-Dwrite=SOCKSwrite \
-Drresvport=SOCKSrresvport \
-Dshutdown=SOCKSshutdown \
-Dlisten=SOCKSlisten \
-Dclose=SOCKSclose \
-Ddup=SOCKSdup \
-Ddup2=SOCKSdup2 \
-Dfclose=SOCKSfclose \
-Dgethostbyname=SOCKSgethostbyname
```

- Link with the library of SOCKS v5 Reference Implementation.

- To the command that creates the executable, add:

```
-L<socks_lib_dir> -lsocks5
```

Using runsocks

Runsocks is a shell script that loads the shared library of SOCKS v5 Reference Implementation. If your operating system supports shared libraries, runsocks dynamically allows the application to use the SOCKSified networking function calls in the shared libraries of SOCKS v5 Reference Implementation instead of the standard networking function calls.

If the application is written using WINSOCKS instead of using the BSD socket API there is an extension that can be used: SocksCap. This WinSock extension will SOCKSifies Winsock Applications. This can be downloaded for non profit use at : http://www.socks.nec.com/reference/sockscap.html. Will automatically enables Windows-based TCP and UDP networking client applications to traverse a SOCKS firewall. SocksCap intercepts the networking calls from WinSock applications and redirects them through the SOCKS server without modification to the original applications or to the operating system software or drivers.

3.4 Porting of Java

Java is an object oriented programming language designed to develop Internet applications. It was designed with the security for programming in mind, been the first object oriented language with network support fully integrated into the language core.

One of the key points for the language design is portability and platform independence. This is obtained by a wide use of software patterns, been one of the first Object Oriented programming language which integrate it deeply into its design. The broad and wise use of this patterns is one of the key point from the separation of the implementation details from the objects invocation that will allow the porting to be an easy task. Another outstanding issue for the design was underlying machine and operating system independence. This was obtained by making the binary code machine independent and running it on a virtual machine.

Networking applications used the socket interface implemented inside the virtual machine, well isolated by a use of the proxy, delegation and factory designs patterns.

The language objectives of secure code download and platform neutrality through Internet was fulfilled by using a generic virtual machine code (bytecode) and a Java Virtual Machine that interprets it.

The Java Virtual Machine can link C libraries in order to use some special Native methods to extend the functionality that can not be coded in the Java language for some special reasons, like performance or access to other APIs.

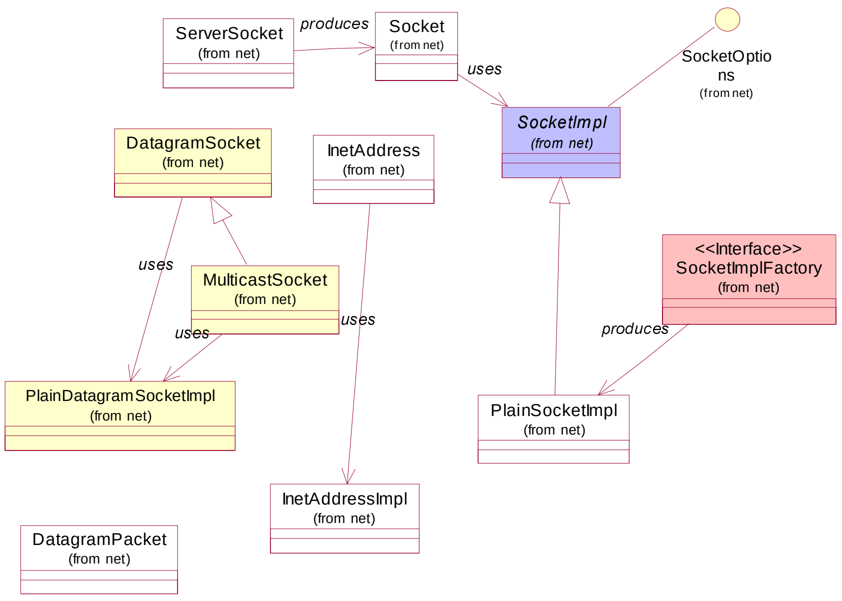

ServerSocket (from net)
produces
Socket (from net)
uses
SocketOptions (from net)
SocketImpl (from net)
DatagramSocket (from net)
InetAddress (from net)
uses
MulticastSocket (from net)
uses
uses
<<Interface>> SocketImplFactory (from net)
produces
PlainDatagramSocketImpl (from net)
PlainSocketImpl (from net)
InetAddressImpl (from net)
DatagramPacket (from net)

Figure 3-8. java.net UML description

The Java network API is the basic package of java.net. This API is partially written in java and partially implemented using native methods. The main classes structure is defined in UML diagram (Figure 3-8). So with this design, it is easy to provide new classes to uses the IPv6 facilities.

3.4.1.1 Jano IPv6 Porting

JANO (Java Advanced NetwOrk facilities) is an open software project which aims to enhance the actual java network API with new facilities in order to design new applications. The project is hosted by Sourceforge (http://sourceforge.net/projects/jano/) which grants support to open software initiatives.

The idea is to design a replacement for the socket building factories to create different sockets that uses IPv6 instead of IPv4 ones. Later this idea will allow the applications to access to new facilities .

By using this porting only some advance communication programs will need to change something in the code. It goal is to be as much transparent as possible to the end programmer and developer.

IPv6 Application porting

Like for the other applications, we can make a taxonomy of porting based on the API parts that the application are using to implement their functionality.

In many cases little work is needed here. Just by modifying the installation of the Java Virtual Machine to load the JANO libraries on the start up, all the sockets factories will be initialized to use the IPv6 implementations. This implies that no change to the code, and even to the bytecode, is needed:

- Since JANO uses the double stack approach, a polymorphism based implementation for both kind of sockets is provided. So the parts of the code that uses hardcoded IPv4 addresses will also work.

- *Standard communications API*: All the HTTP based communication will be built upon the TCP sockets, so it will work without any change.

- This approach will fail if the software made some changes to the underlying factories for sockets. This is done by some software the uses special sockets implementations, like SSL or TSL software. In this case we will need to recode the SSL implementations to work with Ipv6.

- *UDP sockets*: No change needed.

- *Advanced socket use*: Last versions of Java SDK include the possibility of accessing to the raw socket underlying facilities. This is still a work in progress into the JANO porting so, up to now, there is no way of obtaining this service in Java.

The Java security model can limit some of the access to the advance API. It's a good design decision that any code downloaded from the network, like an applet can not access to this. Since the security model change on 1.3 you need to sign and give permissions per function. This affects different API like the multimedia API (JMF: Java Media Framework) to support IPv6. This API implements the RTP and RTSP protocols using actual socket API. In order to port this, a new permissions profile is needed. This is a work in progress into the JANO project.

3.4.1.2 Conclusions

Java application porting to IPv6 is trivial in the 98% of the cases due to the quality of the API design. This API was heavily based on software patterns so reusability is easily achieved.

SUN Microsystems has indicated their intentions to provide IPv6 support in future versions of JDK (1.4) but the time framework is still unclear. The turnaround solution implies using third parties' implementations like JANO.

3.5 Scripting language services

Many of the services that users and administrators use in the internet are not written in a programming language, but prototyped into a scripting language. The origins of scripting languages are traced back to the design of the Unix operating system like a tool box of several cooperating small programmable utilities. Each of the utilities, like awk or sed, has its own small language. This allows the developers to fast obtain the wanted functionality without too much work. This approach has been used for a long time with great success. Up to now several network services are developed using this techniques. One usual software engineering problem with this languages is that they are often "Read Only (ROM)". This means that with a complex syntax, only the person who wrote the program is able to understand and modify it. Even that person has problems after an small amount of time if modifications are required. Instead of been super languages able of solve all the problems in the world, usually a language focuses into solving a problem into a field of computer and networking software engineering science. For an small amount of the languages available you can look at the hello world page (http://www.latech.edu/~acm/HelloWorld.shtml) that shows programs that print Hello World in 204 languages.

The approach of the porting will take two different directions:

- Script porting.
- Run time support porting.

Other possible problem about porting is the licence of the run time support. In the current internet world the successful scripting languages run time support are often written in C language for portability, and are publish with a license (BSD-like or GNU-like) that allows the distribution of the modified code.

3.5.1 Perl language

For Perl is possible to work in both directions defined before. The first option is to migrate scripts using a new communications library. The second option is to migrate the complete run time support.

The porting of Perl scripts can be easily developed with the aid of XS. It is a language used to create an extension interface between Perl and some C library which one wishes to use with Perl. The XS interface is combined with the library to create a new library which can be linked to Perl. An **XSUB** is a function in the XS language and is the core component of the Perl application interface.

The XS compiler is called **xsubpp**. This compiler will embed the constructs necessary to let an XSUB, which is really a C function in disguise, manipulate Perl values and creates the glue necessary to let Perl access the XSUB. The compiler uses **typemaps** to determine how to map C function parameters and variables to Perl values. The default typemap handles many common C types. A supplement typemap must be created to handle special structures and types for the library being linked.

The second alternative is to use a new Perl version with integrated IPv6 support. There is a patch for perl 5.004_04 (ftp://ftp.v6.linux.or.jp/pub/Linux/IPv6/perl/). There is a version for an old Linux Debian distribution. The behavior is not good, with errors during installation and the support is not satisfactory.

4. Example of IPv6 application porting: MGEN

We will illustrate the use of the sockets API with some examples taken from the porting of MGENv6. For more detailed information about socket interface extensions for IPv6 see appendix (annex A and annex B).

MGENv6 uses the following code to open an IPv6 UDP socket and it also sets some multicast options:

```
static int OpenTxSocket( unsigned short *thePort,
unsigned char   theTTL,
struct in6_addr localAddr,
char           *interfaceName)
{
    int fd;
    int on=1;
    unsigned int ifc_index;
    struct sockaddr_in6 serv_addr;
    int TTL;
    int buffer;
    /* Open a socket */
    if ((fd = socket(AF_INET6, SOCK_DGRAM, 0)) < 0)
    {
        perror("MGEN: OpenTxSocket: socket() error");
        return -1;
    }
    /*  assign a name to the socket. The source address can
      be override specifiying a different one with the ON
      command on a flow by flow basis. Each flow can have
      one different source address, overriding each one
      with sendmsg() and in6_pktinfo structure
  */
    memset((char *)&serv_addr, 0, sizeof(struct sockaddr_in6));
    ((struct sockaddr_in6 *)&serv_addr)->sin6_family = AF_INET6;
    ((struct sockaddr_in6 *)&serv_addr)->sin6_addr   = localAddr;
    ((struct sockaddr_in6 *)&serv_addr)->sin6_port   = tons(*thePort);

    if (bind(fd, (struct sockaddr *)&serv_addr,
           sizeof(struct sockaddr_in6)) < 0)
  {
      perror("MGEN: OpenTxSocket: bind() error");
      close(fd);
      return -2;
  }
  /* ... */
  TTL = theTTL;
  if(setsockopt(fd, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
          &TTL, sizeof(TTL)) < 0)
      perror(“MGEN:OpenTxSocket:”
             “setsockopt(IPV6_MULTICAST_HOPS) error”);

  ifc_index= if_nametoindex( interfaceName );
  if (setsockopt(fd, IPPROTO_IPV6, IPV6_MULTICAST_IF,
           &ifc_index, sizeof(ifc_index)) < 0)
      perror(“MGEN:OpenTxSocket:”
             “setsockopt(IPV6_MULTICAST_IF) error”);

  return fd;
}  /* end OpenTxSocket() */
```

4.1 Configuring reception socket

To receive IPv6 packets, the application uses the function listed below. Note, MGENv6 defines IPv6 socket options to force the reception of IPv6 packet options:

```
/* enable the socket to receive extension headers...*/
static int OpenRxSocket(unsigned short *port, int BIND)
{
 int fd;
 int  on = 1;
 struct sockaddr_in6 serv_addr;
 char rbuf[16];
 int optlen = 16;
 const unsigned long rbufSize = 122912;
 /* Open a socket */
 if ((fd = socket(AF_INET6, SOCK_DGRAM, 0)) < 0)
 {
     perror("DREC: OpenRxSocket: socket() error");
     return -1;
 }
 if (BIND)
 {
     memset((char *)&serv_addr, 0, sizeof(struct sockaddr_in6));
     ((struct sockaddr_in6 *)&serv_addr)->sin6_family = AF_INET6;
     ((struct sockaddr_in6 *)&serv_addr)->sin6_addr = in6addr_any;
     ((struct sockaddr_in6 *)&serv_addr)->sin6_port = htons(*port);
     if (bind(fd, (struct sockaddr *)&serv_addr,
              sizeof(struct sockaddr_in6)) < 0)
     {
         perror("DREC: OpenRxSocket: bind() error");
       close(fd);
       return -2;
     }
     /* Try set recv socket buffer to a good size */
     memcpy(rbuf, &rbufSize, optlen);
     if (setsockopt(fd, SOL_SOCKET, SO_RCVBUF, rbuf, optlen) < 0)
     {
         /* Ignore errors for now… */
     }
     /* enable the socket to received extension headers...*/
#ifdef IPV6_RECVHOPOPTS
     setsockopt(fd, IPPROTO_IPV6, IPV6_RECVHOPOPTS, &on,
                sizeof(on));
     setsockopt(fd, IPPROTO_IPV6, IPV6_RECVRTHDR,   &on,
                sizeof(on));
     setsockopt(fd, IPPROTO_IPV6, IPV6_RECVDSTOPTS, &on,
                sizeof(on));
#else
     setsockopt(fd, IPPROTO_IPV6, IPV6_HOPOPTS,     &on,
                sizeof(on));
     setsockopt(fd, IPPROTO_IPV6, IPV6_RTHDR,       &on,
                sizeof(on));
     setsockopt(fd, IPPROTO_IPV6, IPV6_DSTOPTS,     &on,
                sizeof(on));
#endif
 }
 return fd;
}  /* end OpenRxSocket() */
```

4.2 Joining/leaving multicast groups

The following two functions are used to join/leave IPv6 multicast groups:

```
static int JoinGroup(struct in6_addr group_addr,
              struct in6_addr iface_addr,
              DrecSocket     *recvSocketList)
{
    /* ... */
    struct ipv6_mreq mreq;
    char aux[INET6_ADDRSTRLEN];
    /* ... */
    mreq.ipv6mr_multiaddr = group_addr;
    mreq.ipv6mr_interface = if_nametoindex( interfaceName );
#ifdef IPV6_JOIN_GROUP
    if( setsockopt(nextSocket->fd, IPPROTO_IPV6,
                   IPV6_JOIN_GROUP, (char *)&mreq,
                   sizeof(mreq))<0)
#else
    if (setsockopt(nextSocket->fd, IPPROTO_IPV6,
                   IPV6_ADD_MEMBERSHIP,(char *)&mreq,
                      sizeof(mreq))< 0)
#endif
    {
      perror("DREC: Error joining multicast group");
      inet_ntop(AF_INET6, &group_addr, aux,
                  INET6_ADDRSTRLEN );
      fprintf(stderr, "DREC: Group:%s\n", aux );
      return FALSE;
    }

    /* ... */

}  /* end JoinGroup() */

static void LeaveGroup(struct in6_addr group_addr,
               struct in6_addr iface_addr,
               DrecSocket     *recvSocketList)
{
    /* ... */
    struct ipv6_mreq mreq;
    char aux[INET6_ADDRSTRLEN];
    /* ...*/

    mreq.ipv6mr_multiaddr     = group_addr;
    mreq.ipv6mr_interface     = if_nametoindex( interfaceName );

#ifdef IPV6_LEAVE_GROUP
    if(setsockopt(theSocket->fd, IPPROTO_IPV6,
                  IPV6_LEAVE_GROUP,(char *)&mreq,
                     sizeof(mreq))<0)
#else
    if(setsockopt(theSocket->fd, IPPROTO_IPV6,
                    IPV6_DROP_MEMBERSHIP,(char *)&mreq,
                    sizeof(mreq))< 0)
#endif
```

```
        {
            perror("DREC: Error leaving multicast group");
            inet_ntop( AF_INET6, &group_addr, aux, INET6_ADDRSTRLEN);
            printf("DREC: Group:%s\n", aux );
        }

    }  /* end LeaveGroup() */
```

In these functions, it is showed the use of different constants for join/leave multicast groups, the use of `inet6_ntop()` to print an IPv6 address, and the form in which the parameters for the IPv6 group are passed through the `ipv6_mreq` structure with the help of function `if_nametoindex()`.

4.3 Sending routing headers

To send routing headers, the API define three primary functions:

```
inet6_rthdr_space()
inet6_rthdr_init()
inet6_rthdr_add()
```

Some other functions are defined and available, but they are not used by MGENv6. The following function is used to send routing headers:

```
/* global variable */
msghdr msg;
void SendRoutingHeader( MgenEvent *nextEvent )
{
  int i;
  struct cmsghdr  *cmsg_ptr;

  /* reserve memory for the cmsghdr and initialize
     the fields for Routing Header */

  cmsg_ptr = inet6_rthdr_init( msg.msg_control + msg.msg_controllen,
                               IPV6_RTHDR_TYPE_0 );

  /* add the intermediate IPv6 address to the structure cmsghdr */

  for( i=0; i<nextEvent->routing.ndirs; i++)
    inet6_rthdr_add(cmsg_ptr, nextEvent->routing.dirs + i,
                    IPV6_RTHDR_STRICT);

  /* set the last hop behavior */

  inet6_rthdr_lasthop(cmsg_ptr, IPV6_RTHDR_LOOSE);
```

```
    /* upgrade the msg_controllen. */

    msg.msg_controllen =
         msg.msg_controllen +
         inet6_rthdr_space(IPV6_RTHDR_TYPE_0,
                           nextEvent->routing.ndirs );
  }
```

The `inet6_rthdr_init()` function initialises the buffer pointed to by `msg.msg_control` to contain a routing header of the specified type. The caller must allocate the buffer, with a size that can be determined by calling `inet6_rth_space()`. The function returns a pointer to the `cmsghdr` structure that contains the routing header option.

The `inet6_rthdr_add()` function adds the IPv6 address to the end of the routing header being constructed.

Finally, the type of source routing is specified for the last hop (only defined in [RFC2292]), and the controllen member is updated with the function inet6_rthdr_space().

4.4 Sending hop by hop and destination options

All hop-by-hop options must be specified by a single ancillary data object. The option is normally constructed using inet6_option_init() and inet6_option_append() functions, define in [RFC2292].

MGENv6 uses the following function to send hop-by-hop options:

```
  void SendOptionHeader( MgenEvent *nextEvent )
  {
    int i;
    struct cmsghdr  *cmsg_ptr;

    /* ... */

    /* This call builds the cmsghdr structure in the control buffer.*/
    if (inet6_option_init(msg.msg_control + msg.msg_controllen,
                        &cmsg_ptr,
                         nextEvent->options_buf[i].type_anci )==-1)
    {
        perror("inet6_option_init:\n");
        exit(1);
    }

    /* the values for the option have been filled  before:
          option.type    = type_of_the_option;
```

```
            option.len     = len_of_data + alignY ;
            option.value[] = data;
         Also, it should be set the values for:
            option.alignX = align X in "xn + y",
                            specific for the option.
            option.alignY = align Y in "xn + y",
                            specific for the option.
      */

      /* this call appends the Hop-by-Hop option */
      /* into the cmsgptr ptr that has been initialized by */
      /* inet6_option_init(). The length of the option is taken */
      /* from option.len value. */

      if( inet6_option_append(cmsg_ptr,
                              &(nextEvent>options_buf[i].type),
                     nextEvent->options_buf[i].alignX,
                              nextEvent->options_buf[i].alignY)==-1)
      {
          perror("inet6_option_append:\n");
          exit(1);
      }

      /* Upgrade msg_controllen. 2 is for type & len values */
      /* of the option. */

      msg.msg_controllen = msg.msg_controllen +
           inet6_option_space( nextEvent->options_buf[i].len + 2 );
    }
```

The inet6_option_init() function is called once per ancillary data object that will contain either hop-by-hop or destination options. It returns 0 on success or -1 on an error.

The first parameter of inet6_option_init() function is a pointer to a previously allocated space that will contain the ancillary data object (a cmsghdr structure). It must be large enough to contain all the individual options to be added by later calls to inet6_option_append().

The second one is a pointer to a pointer to a cmsghdr structure. This pointer is initialized by this function to point to the cmsghdr structure constructed by this function in the buffer pointed by the first argument.

The last argument is either IPV6_HOPOPTS or IPV6_DSTOPTS. This type is stored in the cmsg_type member of the cmsghdr structure pointed to by the first parameter.

The inet6_option_append() appends a hop-by-hop option or a destination option into an ancillary data object that has been initialized by inet6_option_init(). This function returns 0 if it succeeds, or -1 on an error.

The first parameter of inet6_option_append() function is a pointer to the cmsghdr structure that must have been initialized by inet6_option_init().

The next parameter is a pointer to the 8-bit option type. It is assumed that this field is immediately followed by the 8-bit option data length field, which is then followed immediately by the option data. The caller initializes these three fields (the type-length-value, or TLV) before calling this function.

The rest of parameters are the value x and y in the alignment term "xn + y" previously described.

Finally, MGENv6 uses the inet6_option_space() function to upgrade the controllen field.

4.5 Sending source address

CMSG macros are used to access the fields of cmsg header. MGENv6 use the following function:

```
void SendSourceAddress( MgenEvent *nextEvent )
{
  struct cmsghdr *cmsg_ptr;

  cmsg_ptr= (struct cmsghdr *)(msg.msg_control +
                               msg.msg_controllen);

  cmsg_ptr->cmsg_level = IPPROTO_IPV6;
  cmsg_ptr->cmsg_type= IPV6_PKTINFO;
  cmsg_ptr->cmsg_len = CMSG_LEN( sizeof(struct in6_pktinfo) );
  memcpy( CMSG_DATA( cmsg_ptr ), &nextEvent->sourceAddr,
          sizeof(struct in6_pktinfo) );

  msg.msg_controllen= msg.msg_controllen +
                      CMSG_SPACE( sizeof(struct in6_pktinfo) );
}
```

4.6 Receiving extension headers

To receive and interpret extension headers, the following loop is used to parse the `cmsghdr` structure:

```
  /* scan the Ancillary Data Objects... */
  for (cmsg_ptr = (struct cmsghdr *)CMSG_FIRSTHDR(msg);
       cmsg_ptr != NULL;
```

```
        cmsg_ptr = (struct cmsghdr *)CMSG_NXTHDR(msg, cmsg_ptr))
   {
       if (cmsg_ptr->cmsg_len == 0)
       {
           /* Error handling */
           fprintf(stderr, "Error handling with msghdr\n");
           break;
       }

       /* follow with the parse of extension headers */

       /* ... */
   }
```

Next, inside the loop, the `cmsg_prt->cmsg_level` and `cmsg_ptr->cmsg_type` are examined:

```
   /* Parse an IPv6 Routing Header */
     if( (cmsg_ptr->cmsg_level == IPPROTO_IPV6) &&
         (cmsg_ptr->cmsg_type == IPV6_RTHDR) )
     {
         ptr = CMSG_DATA(cmsg_ptr);
         /* process data pointed to by ptr */
         write_routing_header( (struct ip6_rthdr0 *)ptr );
     }
```

To parse hop-by-hop and destination options, the following code applies (inside we can find the loop described before):

```
   /* Parse an  Hop-by-Hop Options */
     if( (cmsg_ptr->cmsg_level == IPPROTO_IPV6) &&
         (cmsg_ptr->cmsg_type == IPV6_HOPOPTS) )
     {
         while (inet6_option_next(cmsg_ptr, &ptr) == 0)
         {
              if (ptr[0] == IP6OPT_ROUTER_ALERT)
             printf("Router Alert option\n");
        else
                   if (ptr[0] == IP6OPT_JUMBO)
                  printf("Jumbo payload option\n");
                   else
             printf("unknown hop-by-hop option type\n");
         }

         if (ptr != NULL)
```

```
            perror("error encountered by inet6_option_next():\n");
        }
```

In BSD, the parsing of hop-by-hop and destination options is made with a different API, [draft-ietf-ipngwg-rfc2292bis-02.txt] instead of [RFC2292], and can be performed as follows:

```
        /* Parse an  Hop-by-Hop Options */
        if( (cmsg_ptr->cmsg_level == IPPROTO_IPV6) &&
            (cmsg_ptr->cmsg_type == IPV6_HOPOPTS) )
        {
            extbuf= (struct ip6_hbh *)CMSG_DATA(cmsg_ptr);
            extlen= (extbuf->ip6h_len + 1) * 8;
            currentlen= 0;
            while( (currentlen= inet6_opt_next(extbuf, extlen,
                                                   currentlen,
                                    &type, &len,
                                                   &databuf)) >= 0 )
            {
              if(type == IP6OPT_ROUTER_ALERT ){
                  uint16_t value;
             inet6_opt_get_val((void *)databuf, 0, &value,
                                        sizeof(value) );
             /* value has the first 4-bytes of data of
                      the option */
                } else
             if (type == IP6OPT_JUMBO)
                 {
                     uint32_t value;
                 inet6_opt_get_val((void *)databuf, 0, &value,
                                            sizeof(value) );
                 /* value has the first 4-bytes of data of
                          the option */
                 } else
                     printf("unknown hop-by-hop option type\n",
                              type, len );
             }
          }
```

Note that there is a new function, inet6_get_val(), and a redefinition of inet6_option_next() with a new list of parameters and a new name: inet6_opt_next().

The inet6_opt_next() function parses the received option extension headers, returning a pointer to the next option. The first two parameters identify the memory zone in which the option can be found, and its size (equivalent to the cmsg_data field of the cmsghdr structure).

The next one specifies the position to continue the scan of the extension buffer (its an offset), and it should either be zero (for the first option) or the length returned by a previous call to inet6_opt_next() (since this function returns the offset that can be used for pointing to the next option). The type/length/value fields are obtained in the next three arguments. If there are not more options, or if the option extension header is malformed, the return value is –1.

The inet6_opt_get_val() function interprets one value element contained in the value field (we can find several elements in a given value field: for instance, several directions in a routing header). This function eases the identification of the data items contained in the value field, that can be of various sizes (1,2,4, or 8 bytes), in the data portion of the option. It requires an offset to point where the value should be extracted, that is updated on each call, to be able to process iteratively a value field; the first item after the option type and length is accessed by specifying an offset of zero. The next parameter points to a buffer where it is saved the extracted data item, and the last one is the length of the buffer. The function returns the offset for the next field which can be used when extracting option content with multiple fields.

5. New applications development

The migration period will be long (some years). Therefore, it will be necessary to take into account what to do with the development of new applications. New applications should be written for IPv6 in top of the IPv6 API. However, support for IPv4 should be considered to work properly not only on dual-stack environments but also on IPv4 enabled interfaces.

There is a need to develop code that could work with the IPv6 interface and with systems that only implement the IPv4 interface. Some code can be added to detect automatically which kind of interface should be used for communications.

```
int ipv6_supported (void)
{
    int s;
#ifdef AF_INET6
    s= socket(AF_INET6, SOCK_STREAM, 0);
    if (s != -1) {
        (void) close(s);
        return (1);
    }
    switch (errno) {
    case EMFILE: /* can´t tell if IPv6 is supported */
        return (-1);
    default: return (0);
    }
#else
    return (0);
#endif
}
```

However, the above code does not handle the case of library symbols that only exist in the IPv6 versions of the library such as `inet_pton(3N)`. Thus the application would have to use dlsym(3X) with RTLD_NEXT to access these symbols and relay just on IPv4 if the symbols are not available.

6. Evaluation of services and scenarios

The successful deployment of IPv6 over real corporative networks depends on user acceptability where performance is a important criteria. This chapter introduces the performance analysis criteria to take in account to make correct evaluation. The objective is to analyze and study the impact of transitions mechanisms on application performance in terms of bandwidth.

The main characteristics to take in account to compare IPv4 and Ipv6 scenarios are the following:

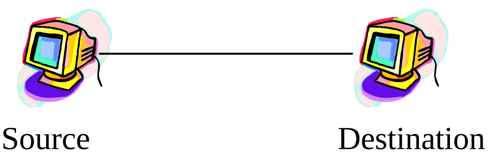
- The packets header size.
- The packets payload. The actual TCP payload is 1460 bytes in IPv4 and 1440 bytes in IPv6.
- Time taken to process IP packets.

The results depends very much on the application protocol. However, the overall performance is estimated about 10% deterioration in IPv6, compared to the IPv4 version.

6.1 Stress tests

First of all, the conditions in which the test is performed should be stated. This should include at least:

- **Hardware:** processor type and speed, memory, and network interface model
- **Software:** operating system version, and patches, service packs or other software that could be required for making the tests. It should also be detailed the steps followed to configure properly the equipment involved in the test. And to fully characterize the test conditions, an IP address map could be depicted, and network status and routing table should be listed
- **Topology:** although in the case considered is almost straightforward, the topology should be detailed if additional equipment (hubs for example) is used. The system's general structure could be:



6.1.1 Tests definition

The main object of these tests is to check the behaviour of the system under different load conditions. We are more interested in characterizing UDP behaviour than TCP, since multimedia applications (and most applications requiring QoS guarantees) are based on UDP. However, TCP behaviour will be addressed. The tools employed for the tests are Netperf Mgen6 and Delay6. A brief description of Mgen6 and both can be found in Appendix A, and B, respectively. The parameters to analyse are the throughput that can be achieved, delay, and jitter.

6.1.2 TCP behaviour

For the analysis of TCP behaviour Netperf will be used. Netperf (www.netperf.org) has been ported to IPv6. The test to perform is to execute netserver on one machine, and netperf –H serverName on the other. A transmission test is carried out for 10 seconds, trying to achieve maximum performance; the throughput obtained is the merit figure of the test.

6.1.3 UDP Rate-limit

We can use the traffic generator tool Mgen adapted to IPv6 to find out the maximum rate a system can support. All we have to do is increase monotonically the rate we send to the destination. This experiment should be performed for different packet sizes.
We can try with packet sizes (in bytes) of 16, 64, 128, 512, 1024, 1408. For each packet size, a series of experiments should be carried out, in which the number of packets/s sent is varied. We should perform at least 20 tests per packet size (in order to be able to obtain enough number of results), so we could divide the total bandwidth available for a given interface into 20 to obtain the increase in bandwidth we should apply from test to test. For example, if we are testing a 100 Mbits/s, we could increase the packet sending rate in steps of 100 Mbits/s / 20 = 5 Mbits/s (the correspondent number of packets/s should be computed, taking into account the packet size, and given as a parameter to the mgen tool); therefore, for a given test serie – for a given packet size, we can begin with 5 Mbits/s with increases of 5 Mbits/s on each test until 100 Mbits/s is reached.

The result could be a graphic with the cursed rate in the “y” axis and the packet size (or kbps generated for every packet size) in the “x” axis. Different lines represent different packet sizes.

6.1.4 UDP Delay

It is possible to use the `mgen` program to find the delay a packet experiments from the source to the destination, but an accurate synchronization between machines is required. To overcome this problem, we can use the Delay6 tool to estimate the delay from the round-trip time measured.

The most interesting parameter to vary is the delay between the generation of packets, in order to obtain as results both main an maximum delays. We can use the same methodology described for the throughput analysis: make several series for different packet sizes, varying on each serie the sending rate in 1/20th of the maximum rate intervals. Note that in this case there are two flows, one from source to echo server, and other from echo server to source; so being different scenarios, the results cannot be directly linked to those obtained in the throughput test. The result will be a graphic with the mean delay in the “y” axis and the throughput generated in the “x” axis.

6.1.5 UDP Jitter

One of the results the Mgen program provides is the mean delay between packet arrival (it is supposed that the generation rate is constant), as well as the maximum and minimum variation in the delay between packets. We can use these measures to find the jitter in the system.

We can use the same methodology described for the previous analysis: make several series for different packet sizes, varying on each serie the sending rate in the same rate intervals presented. The results will be a graphic representing the mean jitter in the "y" axis versus the generated rate in the "x" axis.

6.1.6 Further considerations

First of all, the experiments have been described for IPv6 systems, but all of them should also be tested with IPv4, and the results compared.

We should also make cross experiments involving different operating systems (however, only between those OS with an Mgen port, Linux and Free-BSD – unfortunately, it has not been ported to the Windows platform). For Delay, a Linux PC is required in the origin, but the destination can be any "echo" server, so Linux source and FreeBSD destination can be tested.

7. Conclusion

The migration of applications is not too much difficult. It can be made in parallel with network migration and after the provision of a DNS service (and maybe a NIS service).

It is clear that the ability for a particular portion of code to be ported to IPv6 depends on its own context and the process can not be defined rigidly. However, it can be safely said that, for simple networking interface, the porting is quite straightforward and, sometimes can be automated. Applications with changes in the interface or addition of new functionalities require too much effort, forcing the programmer to analyse and rewrite more than the code that access to the simple sockets interface.

8. Glossary and Abbreviations

API	Advanced Programming Interface.
BSD	Berkeley Software Distribution.
DLL	Dinamic Link Library.
DNS	Domain Name System.
GPL	General Public License.
ICMP	Internet Control Message Protocol (ICMPv4, ICMPv6).
IP	Internet Protocol (IPv4, IPv6).
ISDN	Integrated Services Data Network.
QOS	Quality of Service.
RFC	Request For Comments.
TCP	Transmision Control Protocol.
UDP	User Datagram Protocol.
UML	Unified Modeling language.
VPN	Virtual Private Network.
Double stack	Two interfaces IPv4 and IPv6 are available simultaneously.

9. References

[RFC1881] *IPv6 Address Allocation Management*. IAB, IESG. December 1995. (Format: TXT=3215 bytes) (Status: INFORMATIONAL)

[RFC2292] *Advanced Sockets API for IPv6*. W. Stevens, M. Thomas. February 1998. (Format: TXT=152077 bytes) (Status: INFORMATIONAL)

[RFC2375] *IPv6 Multicast Address Assignments*. R. Hinden, S. Deering. July 1998. (Format: TXT=14356 bytes) (Status: INFORMATIONAL)

[RFC2553] *Basic Socket Interface Extensions for IPv6*. R. Gilligan, S., Thomson, J. Bound, W. Stevens. March 1999. (Format: TXT=89215 bytes) (Obsoletes RFC2133) (Status: INFORMATIONAL)

[RFC2732], *Format for Literal IPv6 Addresses in URL's*. R. Hinden, B., Carpenter, L. Masinter. December 1999. (Format: TXT=7984 bytes) (Status: PROPOSED STANDARD)

[RFC2894] *Router Renumbering for IPv6*. M. Crawford. August 2000. (Format: TXT=69135 bytes) (Status: PROPOSED STANDARD)

[draft-ietf-ipngwg-rfc2292bis-02.txt] W. Stevens, M. Thomas, E. Nodmark . *Advanced Sockets API for IPv6*. November 2000.

[URL’s draft-ietf-ipngwg-url-literal-02.txt] *Preferred Format for Literal IPv6 Addresses* (July 14, 1999)

[draft-ietf-ngtrans-socks-gateway-04.txt] *A SOCKS-based IPv6/IPv4 Gateway Mechanism*, 04/06/2000.

[draft-ietf-ngtrans-translator-03.txt] *Overview of Transition Techniques for IPv6-only to Talk to IPv4-only Communication*, 03/09/2000.

[draft-ietf-ipngwg-rfc2553bis-00.txt] *Basic Socket Interface Extensions for IPv6*, 05/11/2000.

[draft-ietf-ipngwg-rfc2292bis-01.txt] *Advanced Sockets API for IPv6*, 10/28/1999.

[stevens1] Richard Stevens, *Unix Network Programming – Volume 1*

[stevens2] Richard Stevens, Gary Wright, *TCP/IP Illustrated – Volume 2*

http://msdn.microsoft.com/library/psdk/winsock/apistart_9g1e.htm

http://msdn.microsoft.com/downloads/sdks/platform/tpipv6/faq.asp

http://www.sockets/winsock2.htm and http://www.stardust.com/winsock/

A. Appendix A: Basic socket interface extensions for IPv6

This appendix is aimed to describe the socket interface extensions to support IPv6. 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 [RFC2292].

A.1 New Structures

The protocol family in the socket API defines the domain in which communications take place. IPv4 protocols use the `PF_INET` protocol family in the socket API. A new protocol family is defined for IPv6 protocols, `PF_INET6`. There is also a new address family name, `AF_INET6`. The `AF_INET6` definition distinguishes between the original `sockaddr_in` address data structure, and the new `sockaddr_in6` data structure. The following `sockaddr_in6` structure holds IPv6 addresses for 4.3 BSD based systems and is defined as a result of including the `<netinet/in.h>` header (using Posix.1g definition):

```
struct sockaddr_in6
{
   sa_family_t     sin6_family;   /* AF_INET6 */
   sin_port_t      sin6_port;     /* transport layer port # */
   uint32_t        sin6_flowinfo; /* IPv6 traffic class &
                                     flow info*/
   struct in6_addr sin6_addr;     /* IPv6 address */
   uint32_t        sin6_scope_id; /* set of interfaces for a
                                     scope*/
};
```

4.4 BSD based systems define another sockaddr_in6 structure incompatible with 4.3 BSD variant:

```
struct sockaddr_in6
{
   uint8_t         sin6_len;      /* length of this struct */
   sa_family_t     sin6_family;   /* AF_INET6 */
   sin_port_t      sin6_port;     /* transport layer port # */
   uint32_t        sin6_flowinfo; /* IPv6 traffic class &
                                     flow info*/
   struct in6_addr sin6_addr;     /* IPv6 address */
   uint32_t        sin6_scope_id; /* set of interfaces for a
                                     scope*/
};
```

The IPv6 address is stored in network byte order and the structure implementation is the following:

```
struct in6_addr
{
   union  {
      uint8_t  _S6_u8[16];
      uint32_t _S6_u32[4];
      uint64_t _S6_u64[2];
   } _S6_un;
};
#define s6_addr _S6_un._S6_u8
```

IPv6 applications interoperate with IPv4 applications through the use of IPv4-mapped IPv6 address format. This address format allows the IPv4 address of an IPv4 node to be represented as an IPv6 address and therefore it can be used like any other IPv6 address inside of these structures.

A.2 Socket functions

The only difference between IPv4 and IPv6 socket functions are values which these functions are called.

The socket function is called using `PF_INET` for IPv4 or `PF_INET6` for IPv6 as protocol value. Functions using a socket descriptor should manage format address structures according to its socket protocol family.

A generic socket address, named `sockaddr`, is used when passing arguments of the socket functions between kernel and applications. So, these functions can deal with socket address structures from any other supported families. Applications and the kernel must cast the specific to the generic socket address structure when calling socket API functions. But the programming model is worth the same for UDP and TCP applications over IPv4 and IPv6.

```
struct sockaddr
{
   uint8_t sa_len;
   sa_family_t sa_family;    /* AF_XXX: any of supported
                                    protocol family */
   char    sa_data[14];  /* Address depending on family */
};
```

The wildcard address value typically used in the `bind` function to tell the kernel to choose the source local IP address changes when using IPv6. With IPv4, it is used the `INADDR_ANY`, with IPv6 the system initializes the `in6addr_any` variable to the constant `IN6ADDR_ANY_INIT`, included in `<netinet/in.h>`.

The loopback address value, `INADDR_LOOPBACK`, used to establish IPv4 communications in a local node is modified when using IPv6 protocol. With IPv6 the system initializes the `in6addr_loopback` variable to the constant `IN6ADDR_LOOPBACK_INIT`.

A.3 Interface identification

Some new functions are defined to deal with interface name and interface index (a small positive integer that is used to identify uniquely the local interface). These functions are required, for example, to specify the interface on which a multicast group is joined. The information related to an interface is saved in the `if_nameindex` structure:

```
/* if_nameindex structure hold
struct if_nameindex
{
 unsigned int   if_index;  /* 1, 2, ... */
 char          *if_name;   /* null terminated name: "le0" */
};
```

These are the functions which manage the interface information:

```
/* Convert an interface name to an index, and viceversa */
  char *if_indextoname(unsigned int ifindex, char *ifname);
  unsigned int  if_nametoindex(const char *ifname);

  /* Return a list of all interfaces and their indices. */
  struct if_nameindex  *if_nameindex(void);

  /* Free the data returned from if_nameindex. */
  void  if_freenameindex(struct if_nameindex *ptr);
```

A.4 New socket options

There are new socket options processed by IPv6 and handled at the `IPPROTO_IPV6` level, it is the code in the system to interpret the option. The `IPV6_` prefix is used in all of the new socket options. The Table 9-1 shows the new IPv6 socket options.

Table 9-1. IPv6 socket options

Option	Description
IPV6_UNICAST_HOPS	Controls the hop limit used in outgoing unicast IPv6 packets.
IPV6_MULTICAST_IF	Controls the interface to use for outgoing multicast packets. The argument is the index of the interface to use.
IPV6_MULTICAST_HOPS	Controls the hop limit for multicast IPv6 packets.
IPV6_MULTICAST_LOOP	If a multicast datagram is sent to a group to which the sending host itself belongs, a copy of the datagram is looped back by the IP layer for local delivery if this option is set to 1.
IPV6_JOIN_GROUP	Joins a multicast IPv6 group, for BSD systems.

`IPV6_LEAVE_GROUP`	Leaves a multicast IPv6 group, for BSD systems.
`IPV6_ADD_MEMBERSHIP`	Joins a multicast IPv6 group, for Linux systems.
`IPV6_DROP_MEMBERSHIP`	Leaves a multicast IPv6 group, for Linux systems.

A.5 Address resolution and handling

Two functions are used to resolve names: `gethostbyname()` and `gethostbyname2()`. These functions search for the resolution in local files, such as /etc/hosts, and in the DNS system. The query result could be an IPv4 and/or an IPv6 address, it on a variable called `RES_USE_INET6`. Its value is 1 when it is active and 0 when not. This value can be changed in three ways:

- **For only one application:** it is modified by calling the `res_init()` function and including the following option:

```
# include <resolv.h>
void myFunction () {
    /* ... */
    res_init();
    res.options | =RESUSE_INET6;
    /* ... */
}
```

- **For a particular environment:** `RES_USE_INET6` is active if the environment variable `RES_OPTIONS` equals to `inet6`:

```
root@host> export RES_OPTIONS=inet6
```

- **For a whole host:** Only one line must be added to the `/etc/resolv.conf` file:

```
Options inet6
```

The `gethostbyname()` function, included in `<netdb.h>`, returns a pointer to a structure called `hostent` which contains all the IPv4 or IPv6 addresses related to a hostname., depending on the value of the `RES_USE_INET6` variable.

```
struct hostent *gethostbyname (const char *hostname);
```

If `RES_USE_INET6` equals to 0, all the IPv4 addresses are returned. In the other case, if `RES_USE_INET6` equals to 1, IPv6 addresses are returned or those existing IPv4 addresses are returned as IPv4-mapped IPv6 addresses.

The difference between `gethostbyname()` and `gethostbyname2()` is that the latter allows to specify the address family (`AF_INET` for IPv4 and `AF_INET6` for IPv6) to be returned:

```
struct hostent *gethostbyname2 (const char *hostname, int family);
```

There is another function `getipnodebyname()` which allows to select the type of returned address using `flags` argument:

```
        struct hostent *getipnodebyname (char *name, int af,
                                         int flags, int *ernum);
```

All previous functions are dependent on network protocol. There is a way to make the same translation as protocol independent operation using the `getaddrinfo()` function. Developers should use this function instead of previous ones in order to make protocol independent source code which could be easily ported to other network platforms.

```
int getaddrinfo(const char *nodename, const char *servname,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

The `addrinfo` structure is defined as a result of the inclusion of the `<netdb.h>` header and contains all information to be used in creating a socket from a network end point:

```
    struct addrinfo
    {
        int     ai_flags;     /* AI_PASSIVE, AI_CANONNAME,
                      AI_NUMERICHOST */
        int     ai_family;    /* PF_xxx */
        int     ai_socktype;  /* SOCK_xxx */
        int     ai_protocol;  /* 0 or IPPROTO_x for IPv4 and IPv6 */
        size_t  ai_addrlen;   /* length of ai_addr */
        char   *ai_canonname; /* canonical name for nodename */
        struct sockaddr  *ai_addr; /* binary address */
        struct addrinfo  *ai_next; /* next struct in linked list */
    };
```

The `addrinfo` structure returned by `getaddrinfo()` should be deallocated using `freeaddrinfo()` function:

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

The `nodename` and `servname` arguments of `getaddrinfo()` function are pointers to null-terminated strings or to null. One, or both of these two arguments must be a non-null pointer. A non-null `nodename` string can be either a node name or a numeric host address string.

The caller can optionally pass an `addrinfo` structure, pointed by the third argument, to provide hints concerning the type of socket that the caller supports. In this hints, structure members other than `ai_flags`, `ai_family`, `ai_socktype`, and `ai_protocol` must be zero or a null pointer. A value of `PF_UNSPEC` for `ai_family` means the caller will accept any protocol family. A value of 0 for `ai_socktype` means the caller will accept any socket type. A value of 0 for `ai_protocol` means the caller will accept any protocol.

This `getaddrinfo()` function returns a pointer to a linked list of one or more `addrinfo` structures through the final argument. The caller can process each `addrinfo` structure in this list by following the `ai_next` pointer, until a null pointer is encountered.

The following example illustrates the use of `getaddrinfo()` function:

```
bzero(&hints, sizeof(struct addrinfo));

hints.ai_family = PF_INET6;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;

ret_ga = getaddrinfo(argv[0], port, &hints, &res);

if (ret_ga)  err(1, "connect: %s\n", gai_strerror(ret_ga));
if ((s = socket(res->ai_family, res->ai_socktype, 0)) < 0)
    err(1, "socket");
```

The function `gai_strerror()` is defined to aid the applications in printing error messages based on constants returned by `getaddrinfo()`.

Translations from socket address to hostname and service location can be made using `getnameinfo()` function:

```
int getnameinfo(const struct sockaddr *sa, socklen_t salen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

If the returned value is non-zero the resolution succeed.

A.6 Address conversion functions

The addresses are represented in two different ways: the structure used by programs and the ASCII format used to print the addresses so they can easily be understood, named presentation format. There are functions which perform the translation between both kinds of representation addresses:

```
int         inet_pton(int af, const char *src, void *dst);
const char *inet_ntop(int af, const void *src, char *dst,
                      size_t size);
```

The `inet_pton()` function converts an address in its standard text presentation form into its numeric binary form.

The `inet_ntop()` function converts a numeric address into a text string suitable for presentation

In order to allow applications to easily declare buffers of the proper size to store IPv4 and IPv6 addresses in string form, the following two constants are defined in `<netinet/in.h>`:

```
#define INET_ADDRSTRLEN    16
#define INET6_ADDRSTRLEN   46
```

A.7 New macros

There are defined macros for testing special IPv6 addresses. All this macros have the prefix `IN6_IS_ADDR_XXXX`. Next there is a brief list of them:

```
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 *);
```

B. Appendix B: Advanced sockets API for IPv6

This section is based mainly on the specification [RFC2292], that describes the features of IPv6 that some applications will need for managing raw sockets and accessing to IPv6 headers not addressed in RFC-2553. The Linux API is based on [[RFC2292]], while FreeBSD conforms to [draft-ietf-ipngwg-rfc2292bis-02.txt] , a slight variation of the former RFC.

B.1 Accessing to IPv6 and extension headers

Applications and kernel can exchange the following optional information:

- Send/receive interface and source/destination address.
- Hop limit.
- Next hop address.
- Routing header.
- Hop by hop options.
- Destination options.

There are two different mechanisms for the specification of the sending or reception of optional information, based on **`setsockopt()`**, to specify the option content for a socket, named "sticky" options since they affect all transmitted packets on the socket until either a new **`setsockopt()`** is called, or the options are overridden using ancillary data, to specify the option content for a single datagram. This only applies to datagram and raw sockets; not to stream sockets.

B.2 Ancillary data

The most general I/O socket functions are: **`recvmsg()`** and **`sendmsg()`**. Both of them use the **`msghdr`** structure to save information related to the I/O operations. This structure is included in `<sys/socket.h>`:

```
struct msghdr
{
    void        *msg_name;  /* ptr to socket address
                                       structure */
    socklen_t       msg_namelen;   /* size of socket address
                                       structure */
    struct iovec   *msg_iov;   /* scatter/gather array */
    size_t          msg_iovlen;    /* # elements in msg_iov */
    void            *msg_control;   /* ancillary data */
    socklen_t   msg_controllen;/* ancillary data buffer len */
    int      msg_flags;     /* flags on received message */
    };
```

The ancillary data fields of the `msghdr` structure provide a clean way to pass information in addition to the data that is being read or written. The inclusion of the `msg_control` and `msg_controllen` members of the `msghdr` structure along with the `cmsghdr` structure that is pointed to by the `msg_control` member makes possible the specification of ancillary data objects for the send/receive operations performed with UDP sockets.

Ancillary data consist of a list of one or more ancillary data objects, each one beginning with a `cmsghdr` structure:

```
struct cmsghdr
{
    socklen_t  cmsg_len;  /* # bytes, including this
                                      header */
    int     cmsg_level;    /* originating protocol */
    int     cmsg_type; /* protocol-specific type */
    /* followed by unsigned char cmsg_data[]; */
};
```

Hence, `msg_control` points to the first `cmsghdr` structure and `msg_controllen` is the length of all included `cmsghdr`. The following Table B-1 shows macros which simplify the processing of ancillary data objects that a `cmsghdr` structure has as parameters:

Table B-1. Macros to simplify the processing of ancillary data

Macros	Definition
`CMSG_FIRSTHDR()`	Returns a pointer to the first `cmsghdr` in the `msghdr` structure.
`CMSG_NXTHDR()`	Returns a pointer to the `cmsghdr` structure describing the next ancillary data object.
`CMSG_DATA()`	Returns a pointer to the data, named `cmsg_data[]` member.
`CMSG_SPACE()`	Returns an upper bound on the space required by the object and its `cmsghdr` structure, including any padding.
`CMSG_LEN()`	From the length of ancillary data object returns the value to store in `cmsg_len` member of the `cmsghdr` structure.

B.3 Socket options and ancillary data

As we have already seen, to receive optional information applications must select the necessary socket option and must use ancillary data to manage this optional information. The following Table B-2 shows the relation between the socket options and the data type filled in the cmsg_type of the ancillary field.

Table B-2. Relation between socket options and cmsg_type of ancillary data.

[RFC2292] Option name	[draft-ietf-ipngwg-rfc2292bis-02.txt] Option name	Cmsg_type
IPV6_PKTINFO	IPV6_RECVPKTINFO	IPV6_PKTINFO
IPV6_HOPLIMIT	IPV6_RECVHOPLIMIT	IPV6_HOPLIMIT
IPV6_RTHDR	IPV6_RECVRTHDR	IPV6_RTHDR
IPV6_HOPOPTS	IPV6_RECVHOPOPTS	IPV6_HOPOPTS
IPV6_DSTOPTS	IPV6_RECVDSTOPTS	IPV6_DSTOPTS
IPV6_RTHDRDSTOPTS	IPV6_RECVRTHDRDSTOPTS	IPV6_RTHDRDSTOPTS

When any of these options is enabled, the corresponding data is returned as control information by recvmsg(), as one or more ancillary data objects.

B.4 Hop by hop options and destination options

A variable number of hop-by-hop options can appear in a single hop-by-hop option header. Each option in the header is TLV-encoded with a type, length, and value field.

The options have alignment restrictions. The alignment of the first byte of each option is specified by tow values, called x and y, written as “xn+y”. This states that the option must appear at an integer multiple of x bytes from the beginning of the option header (x can have the values 1,2,4 or 8), plus y bytes (y can have a value between 0 and 7, inclusive).

Applications must enable the IPV6_HOPOPTS/IPV6_RECVHOPOPTS socket option to receive hop by hop options. When using ancillary data, cmsg_level must equal to IPPROTO_IPV6 and cmsg_type must be IPV6_HOPOPTS.

Applications must enable the IPV6_DSTOPTS/IPV6_RECVDSTOPTS socket option to receive destination options. When using ancillary data, cmsg_level must equal to IPPROTO_IPV6 and cmsg_type must be IPV6_DSTOPTS.

There are defined four functions to build an option to send:

```
/* returns the number of bytes to hold an option */
int inet6_option_space     (int nbytes);

/* called for each ancillary object which contains a
   hop by hop option or a destination option */
int inet6_option_init      (void *buf, struct cmsghdr **cmsgp,
                            int type);
```

```
    /* appends a hop by hop option or a destination option
       to an ancillary object */
    int inet6_optionappend     (struct cmsghdr *cmsg,
                                const uint8_t *typep,
                                int multx, int plusy);

    /* allocates an ancillary object */
    uint8_t *inet6_option_alloc(struct cmsghdr *cmsg, int datalen,
                                int multx, int plusy);
```

There are defined two functions to process a received option:

```
   /* process the next option */
int inet6_option_next(const struct cmsghdr *cmsg, uint8_t **tptr);

/* search for an option type */
int inet6_option_find(const struct cmsghdr *cmsg, uint8_t *tptrp,
                      int type);
```

B.5 Routing headers

Applications must enable the IPV6_RTHDR/IPV6_RECVRTHDR socket option to receive the routing header. When using ancillary data, cmsg_level must equal to IPPROTO_IPV6 and cmsg_type must be IPV6_RTHDR.

There are defined four options to build one of this kind of options to send:

```
    /* returns number of bytes to hold ancillary data containing
       a routing header */
    size_t inet6_rthdr_space        (int type, int segments);

    /* initialises an ancillary data to contain a routing header */
    struct cmsghdr *inet6_rthdr_init(void *bp, int type);

    /* adds an Ipv6 address to the end of the routing header */
    int inet6_rthdr_add             (struct cmsghdr *cmsg,
                                     const struct in6_addr *addr,
                                     unsigned int flags);

    /* specifies the flag for the final hop */
    int inet6_rthdr_lasthop         (struct cmsghdr *cmsg,
                                     unsigned int flags);
    There are defined three functions to process a received routing header:
      int               inet6_rth_reverse (void *in, void *out);
      int               inet6_rth_segments(const void *bp);
      itruct in6_addr *inet6_rth_getaddr (const void *bp, int index);
```

B.6 Packet info

The source/destination address, the outgoing/arriving interface index, the outgoing/arriving hop limit and the next hop address can be specified using the IPV6_PKTINFO ancillary data object. The structure is:

```
struct in6_pktinfo
{
    struct in6_addr    ipi6_addr; /* src/dst IPv6 address */
    unsigned int   ipi6_ifindex;  /* send/recv interface index */
};
```

Applications must enable the `IPV6_PKTINFO/IPV6_RECVPKTINFO` socket option to receive the packet information. When using ancillary data, cmsg_level must equal to `IPPROTO_IPV6` and `cmsg_type` must be `IPV6_PKTINFO`.

Applications must enable the `IPV6_HOPLIMIT/IPV6_RECVHOPLIMIT` socket option to receive the received hop limit. When using ancillary data, `cmsg_level` must equal to `IPPROTO_IPV6` and cmsg_type must be `IPV6_HOPLIMIT`.

To receive next hop address info, applications must specify in the ancillary data the cmsg_type as `IPV6_NEXTHOP`.

C. Appendix C: Mgen6

To use Mgen6 (a tool adapted to IPv6 by University Carlos III), a traffic emitter (`mgen6`) has to be started in the source terminal and the traffic receiver (`drec6`) in the destination terminal.

So the first thing we have to do is to start the drec program:

```
./drec6  -p 5000 -i eth1 BE1
```

Where we ask the program to listen on interface eth1 and port 5000, and to redirect the results to file BE1.

Next , we have to start the `mgen6` in the traffic source :

```
./mgen6 gscript
```

Where *gscript* is the configuration file to use.

An example coud be:

```
##########################################

INTERFACE eth1

# <time>  <flow_id>  ON <addr:port> <pattern> <rate>  <size>
# time is in miliseconds
# <flow_id> is the flow identifier
# <addr port> destination address and port
# <pattern> PERIODIC or POISSON (burst)
# <rate> packets/s
# <size> size of every packet in bytes


00000   1  ON  2002:A375:8B2C:1E:2C0:26FF:FE10:193F 5000
           PERIODIC 100 1024

      10000   1 OFF

##########################################
```

With this configuration, 800 Kbps traffic are sent for 10 seconds from the source to the destination. The processing of the raw data obtained is done after stopping the `drec` program with the `mcalc6` utility in the form.

```
./mcalc6 BE1
```

We can obtain statistics of the traffic for the receiver, like the mean received rate for this flow, the delay variation or the number of packets dropped. Synchronization between machines is required if the delay is to be accurately measured.

D. Appendix D: Delay6

This tool (developed by Telefonica I+D) sends one UDP packet/s to the “echo” port of the remote host, that it returns back to the origin host, allowing the source to measure the delay.

The syntax of the program is:

```
    ./delay6   –p N  -d delay   –H remote_host    –F output_file
```

An example could be:

```
    ./delay6 –H 2002:A375:8B2C:1E:2C0:26FF:FE10:193F –p 500 \
           –d 1000000 –F  f1
```

where we mean we are sending to the remote host 500 packets, the delay between packets is one second, and the output file is f1. Packet length can be selected with the –L flag.

The processing of the data is done with the program `procesa_stat` in the following form:

```
./procesa_stat  f1
```

For every received packet, there is a line with the output time, the arrival time and the round trip time.

We can take the data to an Excel page to find the mean delay, that is the result considered in the proposed measures:

$$\bar{d} = \frac{1}{N}\sum_{i=1}^{N} d_i$$