Chapter 6.0 IPv6 implementation issues in the x-kernel.

The main goal of this dissertation is to look at the issues involved in implementing IPv6 into the x-kernel and its simulator. Generally the x-kernel plays the role of a test-bed for testing various protocol combinations or to implement newly created protocols into the existing protocol combinations, for an example the implementation of IPv6 in the TCP/IP protocol stack. As stated before, the x-kernel implies minimal object oriented style in implementing protocols, however if a new protocol is to be implemented in an existing set of protocol layers, there has to be some significant changes to all protocol objects that are related to this new protocol.

For this dissertation, the issues are focused on hosts connected either via an Ethernet connection or a Point-to-Point connection. This type of implementation sets the proper environment for doing network simulation in x-kernel's simulator x-sim. The issues that were focused on for the IPv6 implementation and its simulation are based on a network configuration with the most minimal IPv6 specification of RFC2460. RFC2460 basically gives the full specification or semantics with no consideration of the programming language or the type of platform used for implementing IPv6 protocols [8]. As for the issues of simulation, a configuration that only establishes a connection between two hosts with the new IPv6 address architecture is proposed. This new address architecture will be apparent in the simulation. Other capabilities such as routing, fragmentation and other options that are specified in RFC2460 for IPv6 are not included in this review.

The x-kernel originally comes with the source code for IPv4 implementation. These codes can be modified to meet the specific requirements of IPv6 [4]. Thus the issues in implementing IPv6 will be based on these codes and its semantics. Before moving on to the implementation issues, here is a table of main differences in IPv6 module as compared to

IPv4. This table shows six columns that are identified as What, Subject, IPv4, IPv6, S(Same), D(Different) and N(New). Table 6.1 describes the above.

What	Subject	IPv4	IPv6	S	D	N
Base functionality	Header format	variable	fixed		+	
		10 fields	6 fields		+	
		2 address fields.	2 address fields	+		
		checksum	no checksum		+	
		ToS	flows		+	+
		Protocol type	Next header type		+	
		IP options	extension headers		+	+
Addressing and routing	Addressing	unicast broadcast	unicast multicast anycast		+	+
	Routing	CIDR	Providers		+	
		BGP-4	IDRP (BGP-4+)		+	+
		OSPF	OSPF with minimal changes		+	
		RIP	RIP	+		
	ICMP	+IGMP	ICMP		+	
Plug and play	Neighbor discovery	ARP+router discovery	Neighbor Discovery		+	
	Auto configuration	no	yes		+	+

Table 6.1 Main differences between IPv4 as compared to IPv6 [6].

IPv6 header fields as compared to IPv4 header fields have major differences; IPv6 header fields and its description are listed in table 6.2:

Version number	6 (The new IP version)
Flow label	Contents of the packet, priority
Payload length	The number of octets in the payload field (data)
Next header	The relative number to the next header (e.g. TCP, UDP)
Hop limit	Lifetime of the respective packet
Source address	Sender (IP Address)
Destination address	Recipient (IP Address)

Table 6.2 IPv6 header field descriptions table.

6.1 The Protocol vs. the Operating System

There are normally three reasonable ways to add a protocol to an operating system.

(i) Firstly, the protocol can be in a process that is provided by the operating system.

(ii) Secondly, the protocol can be part of the kernel of the operating system itself.

(iii) Thirdly, the protocol can be put in a separate communications processor or front-end machine.

This decision is strongly influenced by details of hardware architecture and operating system design; each of these three approaches has its own advantages and disadvantages. The "process" is the abstraction which most operating systems use to provide the execution environment for user programs. A very simple path for

implementing a protocol is to obtain a process from the operating system and implement the protocol to run in it.

Superficially, this approach has a number of advantages. Since modifications to the kernel are not required, the job can be done by someone who is not an expert in the kernel structure. Since it is often impossible to find somebody who is experienced both in the structure of the operating system and the structure of the protocol, this path, from a management point of view, is often extremely appealing. Unfortunately, putting a protocol in a process has a number of disadvantages, related to both structure and performance.

First, as was discussed above, process scheduling can be a significant source of real-time delay. There is not only the actual cost of going through the scheduler, but the problem that the operating system may not have the right sort of priority tools to bring the process into execution quickly whenever there is work to be done. Structurally, the difficulty with putting a protocol in a process is that the protocol may be providing services, for example support of data streams, which are normally obtained by going to special kernel entry points. Depending on the generality of the operating system, it may be impossible to take a program which is accustomed to reading through a kernel entry point, and redirect it so it is reading the data from a process. The most extreme example of this problem occurs when implementing server telnet. In almost all systems, the device handler for the locally attached teletypes is located inside the kernel, and programs read and write from their teletype by making kernel calls. If server telnet is implemented in a process, it is then necessary to take the data streams provided by server telnet and somehow get them back down inside the kernel so that they mimic the interface provided by local teletypes. It is usually the case that special kernel modification is necessary

to achieve this structure, which somewhat defeats the benefit of having removed the protocol from the kernel in the first place. Clearly, then, there are advantages to putting the protocol package in the kernel. Structurally, it is reasonable to view the network as a device, and device drivers are traditionally contained in the kernel. Presumably, the problems associated with process scheduling can be sidestepped, at least to a certain extent, by placing the code inside the kernel. And it is obviously easier to make the server telnet channels mimic the local teletype channels if they are both realized in the same level in the kernel.

However, implementation of protocols in the kernel has its own set of pitfalls. Firstly, network protocols have a characteristic which is shared by almost no other device: they require rather complex actions to be performed as a result of a timeout. The problem with this requirement is that the kernel often has no facility by which a program can be brought into execution as a result of the timer event. What is really needed, of course, is a special sort of process inside the kernel. Most systems lack this mechanism. Failing that, the only execution mechanism available is to run at interrupt time. There are substantial drawbacks to implementing a protocol to run at interrupt time. First, the actions performed may be somewhat complex and time consuming, compared to the maximum amount of time that the operating system is prepared to spend servicing an interrupt. Problems can arise if interrupts are masked for too long. This is particularly bad when running as a result of a clock interrupt, which can imply that the clock interrupt is masked. Second, the environment provided by an interrupt handler is usually extremely primitive compared to the environment of a process. There are usually a variety of system facilities which are unavailable while running in an interrupt handler.

The most important of these is the ability to suspend execution pending the arrival of some event or message. It is a cardinal rule of almost every known operating system that one must not invoke the scheduler while running in an interrupt handler. Thus, the programmer who is forced to implement all or part of his protocol package as an interrupt handler must be the best sort of expert in the operating system involved, and must be prepared for development sessions filled with obscure bugs which crash not just the protocol package but the entire operating system.

A final problem with processing at interrupt time is that the system scheduler has no control over the percentage of system time used by the protocol handler. If a large number of packets arrive, from a foreign host that is either malfunctioning or fast, all of the time may be spent in the interrupt handler, effectively killing the system. There are other problems associated with putting protocols into an operating system kernel. The simplest problem often encountered is that the kernel address space is simply too small to hold the piece of code in question. This is a rather artificial sort of problem, but it is a severe problem none the less in many machines. It is an appallingly unpleasant experience to do an implementation with the knowledge that for every byte of new feature put in one must find some other byte of old feature to throw out. It is hopeless to expect an effective and general implementation under this kind of constraint.

Another problem is that the protocol package, once it is thoroughly entwined in the operating system, may need to be redone every time the operating system changes. If the protocol and the operating system are not maintained by the same group, this makes maintenance of the protocol package a perpetual headache. The third option for protocol implementation is to take the protocol package and move it outside the

machine entirely, on to a separate processor dedicated to this kind of task. Such a machine is often described as a communications processor or a front-end processor.

There are several advantages to this approach. First, the operating system on the communications processor can be tailored for precisely this kind of task. This makes the job of implementation much easier. Second, one does not need to redo the task for every machine to which the protocol is to be added. It may be possible to reuse the same front-end machine on different host computers. Since the task need not be done as many times, one might hope that more attention could be paid to doing it right. Given a careful implementation in an environment which is optimized for this kind of task, the resulting package should turn out to be very efficient. Unfortunately, there are also problems with this approach. More fundamentally, the communications processor approach does not completely sidestep any of the problems raised above. The reason is that the communications processor, since it is a separate machine, must be attached to the mainframe by some mechanism. Whatever that mechanism, code is required in the mainframe to deal with it. It can be argued that the program to deal with the communications processor is simpler than the program to implement the entire protocol package. Even if that is so, the communications processor interface package is still a protocol in nature, with all of the same structural problems.

There is a way of attaching a communications processor to a mainframe host which sidesteps all of the mainframe implementation problems, which is to use some preexisting interface on the host machine as the port by which a communications processor is attached. This strategy is often used as a last stage of desperation when the software on the host computer is so intractable that it cannot be changed in any way. Unfortunately, it is almost inevitably the case that all of the available interfaces are

totally unsuitable for this purpose, so the result is unsatisfactory at best. The most common way in which this form of attachment occurs is when a network connection is being used to mimic local teletypes. In this case, the front-end processor can be attached to the mainframe by simply providing a number of wires out of the front-end processor, each corresponding to a connection, which are plugged into teletype ports on the mainframe computer. This strategy solves the immediate problem of providing remote access to a host, but it is extremely inflexible. The channels being provided to the host are restricted by the host software to one purpose only, remote login. It is impossible to use them for any other purpose, such as file transfer or sending mail, so the host is integrated into the network environment in an extremely limited and inflexible manner. If this is the best that can be done, then it should be tolerated.

6.2Protocol Layering

The previous discussion suggested that there was a decision to be made as to where a protocol ought to be implemented. In fact, the decision is much more complicated than that, for the goal is not to implement a single protocol, but to implement a whole family of protocol layers, starting with a device driver or local network driver at the bottom, then IP and TCP, and eventually reaching the application specific protocol, such as Telnet, FTP and SMTP on the top. Clearly, the bottommost of these layers is somewhere within the kernel, since the physical device driver for the net is almost inevitably located there. Equally clearly, the top layers of this package, which provide the user his ability to perform the remote login function or to send mail, are not entirely contained within the kernel. Thus, the question is not whether the protocol

family shall be inside or outside the kernel, but how it shall be sliced in two between that part inside and that part outside.

Since protocols come nicely layered, an obvious proposal is that one of the layer interfaces should be the point at which the inside and outside components are sliced apart. Most systems have been implemented in this way, and many have been made to work quite effectively. One obvious place to slice is at the upper interface of TCP. Since TCP provides a bi-directional byte stream, which is somewhat similar to the I/O facility provided by most operating systems, it is possible to make the interface to TCP almost mimic the interface to other existing devices. Except in the matter of opening a connection, and dealing with peculiar failures, the software using TCP need not know that it is a network connection, rather than a local I/O stream that is providing the communications function. This approach does put TCP inside the kernel, which raises all the problems addressed above. It also raises the problem that the interface to the IP layer can, if the programmer is not careful, become excessively buried inside the kernel. It must be remembered that things other than TCP are expected to run on top of IP.

The IP interface must be made accessible, even if TCP sits on top of it inside the kernel. Another obvious place to slice is above Telnet. The advantage of slicing above Telnet is that it solves the problem of having remote login channels emulate local teletype channels. The disadvantage of putting Telnet into the kernel is that the amount of code which has now been included there is getting remarkably large. In some early implementations, the size of the network package, when one includes protocols at the level of Telnet, rivals the size of the rest of the supervisor. This leads to vague feelings that all is not right. Any attempt to slice through a lower layer boundary, for example between internet and TCP, reveals one fundamental problem.

The TCP layer, as well as the IP layer, performs a demultiplexing function on incoming datagrams. Until the TCP header has been examined, it is not possible to know for which user the packet is ultimately destined. Therefore, if TCP, as a whole, is moved outside the kernel, it is necessary to create one separate process called the TCP process, which performs the TCP multiplexing function, and probably all of the rest of TCP processing as well. This means that incoming data destined for a user process involves not just a scheduling of the user process, but scheduling the TCP process first. This suggests an alternative structuring strategy which slices through the protocols, not along an established layer boundary, but along a functional boundary having to do with demultiplexing.

In this approach, certain parts of IP and certain parts of TCP are placed in the kernel. The amount of code placed there is sufficient so that when an incoming datagram arrives, it is possible to know for which process that datagram is ultimately destined. The datagram is then routed directly to the final process, where additional IP and TCP processing is performed on it. This removes from the kernel any requirement for timer based actions, since they can be done by the process provided by the user. This structure has the additional advantage of reducing the amount of code required in the kernel, so that it is suitable for systems where kernel space is at a premium.

6.3 The IPv6 module issues

Further discussion on the IPv6 module will be based on the x-kernel simulator x-sim. The x-sim shares the same implementation environment as the x-kernel. Most of the files in the IPv4 module implementation can be used to implement the new IPv6, but some of the files may need modifications to fit the requirements of IPv6 specification [4]. The

IPv4 module contains a set of *.c and *.h extension files which are the source code for IPv4 implementation in the x-kernel. The following review will first look at the existing IPv4 module in the x-kernel then a review on the proposed IPv6 module based on the latter module. Finally issues on the proposed simulation configuration of IPv6.

6.3.1 The IPv4 module

The main file in IPv4 module implementation that needs to be modified is the *ip.c* file. A complete listing of *ip.c* program file is in *Appendix C*. This file is where protocol objects and session objects are utilized by the program codes. In this file the main application for x-kernel's UPI coding takes place in terms of session object and protocol object implementation. In this file there are twenty-three routines or functions that represent the x-kernel's IP protocol layer. This program file basically initializes and prepares the IPv4 protocol objects and session objects for communication. The C codes in this file can be extended to satisfy the IPv6 implementations with minimal but significant changes. For the IPv4 protocol, its module is contained in the directory */usr/xkernel/simulator/protocol/ip*. The C functions that implement a given protocol can be distributed across multiple *.c files as shown in the list below. The list of files in this directory for the IPv4 module are listed below:

- *ip.c*
- *ip_i.h*
- *iproute.c*
- *ip_control.c*
- *ip_input.c*
- *route.h*
- *ip_frag.c*
- *ip_mask.c*
- *route_i.h*
- *ip_gc.c*
- *ip_rom.c*
- *ip_hdr.c*
- *ip_util.c*

In this directory there are also script files for compiling purposes and the object files for execution, which are not shown in the list above. The other files that are related to this modules are the header files, these files are usually in */xkernel/include/prot* directory. The files that are related with the IPv4 module are listed below.

- *ip.h*
- *ip_host.h*
- *ipsec.h*

The figure below depicts main IPv4 module and its relationship between **.c* files and **.h* files based on the source code

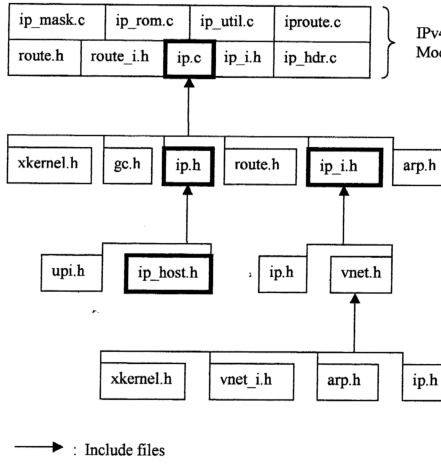

Figure 6.1 The relationship between header files and *ip.c* file in IPv4 module

6.3.2 The proposed IPv6 module

Adding IPv6 capability suggests modifying code to work specifically by adding IPv6 capability while retaining IPv4 functionality. When adding IPv6 functionality, it is necessary to define properly sized data structures. The size of an IPv6 address is much larger than an IPv4 address. Structures that are hard-coded to handle the size of an IPv4 address when storing an IP address will cause problems and must be modified [15]. Based on the C programming language definition; a structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling [11].

The following listing describes the main points of the proposed IPv6 module:

(i) For the implementation of IPv6 module in the x-kernel, first a directory has to be created to save all the necessary files for the build and compiling process. The new directory can be named as *ip6* and it shall be in the */usr/xkernel/simulator/protocol* directory for simulation purposes. The ip6 directory shall contain all the new files for the IPv6 implementation.

(ii) The suitable name for the IPv6's implementation in the x-kernel or x-sim will be *ip6.c*. Further discussion on the proposed issues *ip6.c* will be based on the *ip.c* file of IPv4 module.

(a) As from the *ip.c* file, in it's coding the *createLocalSessn( )* routine plays the function of filling in the session template header with information of IPv4's header fields. Since IPv6 has no header length field, hence this part of the program has to be modified [8].

(b) In the *ipSend( )* routine, for this implementation, as proposed, no fragmentation is considered in IPv6. Thus in this routine all fragmentation part of message is discarded for this implementation. Also in this routine the *hdrlen* variable has to be changed to *payloadLength* as specified in the IPv6 specification.

(c) In the *ipPath( )* routine the macro *GET_HLEN(&hdr) * 4* has to be removed for the IPv6 implementation. As in the IPv6 specification, multicast replaces the broadcast capability of IPv4, so the *fwdBcastSessn( )* routine has to be replaced with a new routine *fwdMcastSessn( )* function.

(d) In the *ipCreatePassiveSessn( )* routine the macro's *BCAST_SUBNET_ADDR_C* and *BCAST_LOCAL_ADDR_C* needs to be changed to the multicast capabilities of IPv6.

(e) The routines *ipRouteChanged( )* and *routeChangeFilter( )* implements the routing aspects for IPv4, these routines also needs to be changed for IPv6, but as for this dissertation it is not taken into consideration because no routing is proposed into the x-kernels simulation.

The main contributors to the changes of IPv6 implementation lie in the IPv4 header files. There are six header files that in *ip.c* file these files are listed below.

- *ip_i.h*
- *ip.h*
- *xkernel.h*
- *arp.h*
- *gc.h*
- *route.h*

(i) From the *ip_i.h* file, the *ip_host.h* file is included, and it is this file, which determines the *iphost* structure for IPv4. This structure basically shows the 32-bit address architecture.

```
typedef struct iphost{
        u_char a, b, c, d;
        }IPhost;
```

To implement IPv6, the structure above needs to change its structure members to accommodate IPv6's 128-bit address architecture. The proposed *iphost6* structure:

```
typedef struct iphost6{
        u_int a, b, c, d, e, f, g, h;
        }IPhost6;
```

This file can be renamed to *ip_host6.h*, with all of the macros in this file also has to be changed and modified to comply with its structure changes.

(ii) Moving on to the next header file, which relates to this discussion is the *ip.h* header file. This header file includes its own header file, which is the *ip_host.h* file, also subsequently requires changes for IPv6. Again the routing aspects in this file are neglected at this time. The structure block *ipsseudohdr* in *ip.h* file needs restructuring for IPv6 and is shown below. The *ipsseudhodr* structure below belongs to IPv4 module. Following this structure block is the proposed structure block for IPv6 and named as *ippseudohdr6*.

typedef struct ippseudohdr {

IPhost src;

IPhost dst;

u_char zero;

u_char prot;

u_short len;

} IPpseudohdr;

The proposed *IPpseudohdr6* structure for *ip6.h* is shown below.

typedef struct ippseudohdr6 {

IPhost6 src;

Iphos6t dst;

u_char zero;

u_char n_header;

u_short payload;

} IPpseudohdr6;

The macros in this file which are *IP_LOCAL_BCAST* has to be changed to *IP_LOCAL_MCAST* and *IP_ADS_BCAST(A)* to *IP_ADS_MCAST(A)* with its respective definition for IPv6 implementation.

(iii)The next header file, which is also included in the *ip.c* file, is the *ip_i.h* file. This header file is one of the most important files in defining the IP header fields. So it is necessary that all files in the new IPv6 module comply and change accordingly based on this file. This file can be renamed as *ip i6.h* to suit IPv6. All the header files in *ip.c* file shall be renamed accordingly to indicate that it is a new file related

with IPv6 module. The *ip.h* and *vnet.h* files renamed and modified as *ip6.h* and *vnet6.h* respectively. The *vnet.h* file is for the simulation purposes. Thus the *arp.h* file is not needed for simulation and IPv6 uses neighbor discovery instead. In fact the ARP module is discarded totally in the IPv6 protocol specification. The important changes in this file is to remove all macros related to *IPHLEN*, *HLEN_MASK*, *VERS_MASK* and the changing of *GET_VERS(h)* relative to IPv6 module. All macros related to fragmentation and routing is also removed. Looking at the main structure that defines the IPv4 header fields is defined in this file, and it is the *IPheader* structure. The proposed structure for IPv6, named as *IPheader6* structure is described below.

```
typedef struct ipheader6{
    u_char vers;
    u_char prio;
    u_char flow;
    u_short payload;
    u_short n_header;
    u_char h_lim;
    IPhost6 source;
    IPhost6 dest;
    } IPheader6;
```

(iv) All the structures and type definitions that use the proposed IPv6 headers need to be modified simultaneously in this module. All structures and routines in this file that relates to fragmentation and routing can be dropped for this implementation.

(v) As for the other files that are included in the *ip.c* file; *arp.h* header file is completely dropped in IPv6, *route.h* is also not considered for this proposed implementation, header files such as *xkernel.h* and *upi.h* remains unchanged. Generally these are the structures and header files in *ip.c* that needs to be changed for the new IPv6 module. For the proposed IPv6 module implementation, all structures, macros, routines and type definitions need to be consistent throughout the whole *ip6.c* file and to all other files that are linked to it.

These are main points to look for in the *ip.c* file to be modified into *ip6.c* for the IPv6 module. This set of changes needs to be consistent towards all **.c* files in the *xkernel/simulator/protocol/ip6* directory for the new *ip6* directory. For this implementation the files listed below are the ones that needs to be considered for modification for newly proposed IPv6 module. These files need to be renamed accordingly for IPv6 module.

- *ip_control.c*
- *ip_gc.c*
- *ip_hdr.c*
- *ip_input.c*
- *ip_rom.c*
- *ip_util.c*

6.4 IPv6 simulation and configuration issues

The new IPv6 module can be easily simulated using the x-kernel's simulator x-sim. This simulator can be configured to just simulate two hosts connected to each other via either an Ethernet or a Point-to-Point connection. Basically in this dissertation, the

simulator is mainly used to show that the new protocol module with its new address architecture can be implemented and tested upon. The new module is to be said operational when there is connection between two virtual hosts. The figure 6.2 show a screen shot of a successful host-to-host simulation in a Linux operating system.

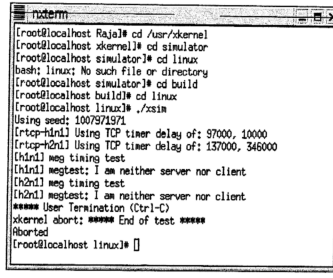

Figure 6.2 A screen shot of IPv4 simulation in progress between two hosts on an Ethernet

This simulator is the safest way to test the new protocols in the user-level before moving the protocol into the x-kernel itself. Further enhancements and changes can be made to the new protocols in the simulator in a much easier and faster way. Other wise, it is not easy compiling and initializing new protocols into an operating system [5]. To test a new protocol in the operating system kernel, it is required to reboot the system each time there is a change made.

Looking at an example from the IPv4 module implementation in the x-sim. All modules that is needed for this simulation is readily available when the whole software was downloaded from the author's website. These tools can be used to emulate the required simulation by configuring the necessary files to suit the users requirements. The figure

below shows the x-sim protocol graph of two virtual hosts and its protocol layers connected by a *SIM* layer, which represents the connection layer or internetwork. In this context the issues related for the IPv6 simulation is based on the IP layer.

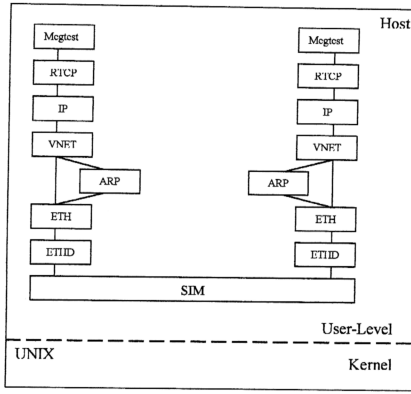

Figure 6.3 Two virtual hosts in a simulation at user-level

Figure 6.3 above shows a protocol graph representing two simulated hosts connected by an internetwork. Protocol layer *SIM* contains code that implements the entire internetwork to which the hosts are attached. In the x-sim, its own protocol stack implements each simulated host, this leads to multiple instantiations of protocols objects, and a large graph.

In a user-level x-kernel, the protocol graph is static and it is specified at compile time in the *graph.comp* file. In contrast, the simulator has been enhanced so that the protocol graph is created at runtime, allowing different simulations to be run without recompiling. The *graph.comp* file is still used by x-sim, but only as a means of specifying which protocol modules to include in the executable. In other words, if any simulated host is going to be running P, then protocol P must appear somewhere in *graph.comp*. The relationships between protocols specified in *graph.comp* are ignored by the simulator.

At runtime, x-sim reads a configuration file, called *xsim.data*, or any other file that can be specified by the *–simfile* option, which specifies the network topology: the number and types of networks; the number of hosts and routers; how the networks, host and routers are connected; the protocol graph for each host; and various other parameters such as the addresses of each host and network, the speed and delay of each Point-to-point link and network, whether Ethernet networks should simulate collisions, and so on. However for the IPv6 simulation issues, parameters such as speed and delay of a Point-to-Point link and network, Ethernet networks with simulated collision and so on is neglected for the time being.

Basically, building the protocol graph for a simulation involves two stages: including the necessary protocol modules in the simulator executables (via *graph.comp*) and specifying at runtime the protocol graphs for the individual simulated hosts(via *xsim.data*).

As for the new IPv6 module, a series of new modules, which is related to, the latter module need to be introduced in the x-sim. This new modules replaces all protocols that are related to the IPv6 module. For the IPv6 layer, some of the IPv4 are modified or removed and new modules added, as to have a full working protocol simulation for IPv6. However

in this context, the issues that was considered was only the IP module or layer from the whole stack of the figure 6.3.

As shown in figure 6.1, the protocol modules that need to be changed are listed in the example *graph.comp* file as shown below. The configuration file *xsim.data* is also needed to be modified for IPv6. An example of this file is also shown below. These files illustrate the basic connection requirements of two isolated hosts.

The proposed *graph.comp* file for IPv6 simulation:

@;

name=sim;

name=ethd;

name=eth;

name=vnet6;

name=ip6;

name=ricp6;

name=megtest6;

@

prottbl=prottbl.default;

prottbl=prottbl.xsim;

The proposed *xsim.data* file for IPv6 simulation:

```
set ip6 = ip6, ip6 vnet6, vnet6 eth, eth ethd, ethd sim;

set buf = 50;

set db = db.out;

set delay = 10ms;

set megtime = 50;

set time = 50;

set rbuf = 15;

# Define hosts

host h1n1;

protocols = megtest6 rtcp6, rtcp6 $ip6;

args = -c2FFF:80::94:1  buf = $buf  -tcpTrace=2000

-db=$db –delay=$delay  -megTime = $megTime;

host h2n1;

protocols = megtest6 rtcp6, rtcp6 $ip6;

args = -c2FFF:80::94:2  buf = $buf  -tcpTrace=2000

-db=$db –delay=$delay  -megTime = $megTime;

# Define network, connectivity

net ETH 2FFF:80::94:0;

connections = h1n1 2FFF:80::94:1,  h2n1  2FFF:80::94:2;

args = rate =200KB, delay = 10ms;
```

6.5 Summary

Finally, as to summarize this chapter, the proposed IPv6 module covers the main issues in implementing IPv6 in the x-kernel. The focus is on the existing *ip.c* file of IPv4 module and its capability to be modified for IPv6. After looking at the main issues of the IPv6 source code, the main factors for a simulation of IPv6 in the x-sim are also considered. The proposed simulator configurations are based on the proposed IPv6 module and other associated protocols in-lieu of the protocol stack in figure 6.3. This proposed simulation shows that this implementation is capable for the simplest virtual host-to-host connection of new IPv6 address architecture.