



681, rue de la Passerelle
Domaine Universitaire
38400 Saint Martin d'Hères



France Télécom R&D – site de Grenoble
28, chemin du Vieux Chêne – BP 98
38243 Meylan Cedex

Third Year Project

Implementation of a network stack for a Java kernel

Denis Carniel, Guillaume Molléda

The 21th of June 2000

Contents

1. ABSTRACT	4
2. FRANCE TÉLÉCOM R&D.....	5
3. AIMS.....	5
4. WORKING ENVIRONMENT.....	6
4.1. THE KERNEL.....	6
4.2. TOOLS.....	6
5. FUNCTIONAL ARCHITECTURE.....	7
5.1. INITIALIZATION	7
5.2. INCOMING MESSAGES	8
5.3. OUTGOING MESSAGE.....	10
6. SOFTWARE ARCHITECTURE	11
6.1. PACKAGES HIERARCHY	11
6.1.1. <i>Network.protocols</i>	11
6.1.2. <i>Network.stack</i>	11
6.1.3. <i>Network.stack.api</i>	12
6.1.4. <i>Network.stack.hardend</i>	12
6.1.5. <i>Network.tools</i>	12
6.2. HOW TO	13
7. IMPLEMENTATION OF THE PROTOCOLS.....	15
7.1. ARP.....	15
7.2. IP.....	15
7.2.1. <i>Protocol and Session objects</i>	15
7.2.2. <i>Fragmentation and reassembly</i>	16
7.2.3. <i>Routing</i>	17
7.2.4. <i>Adding a new type of hardware interface</i>	18
7.2.5. <i>Type of Services and Options</i>	18
7.3. UDP	19
7.4. ICMP.....	19
8. DIFFERENCES BETWEEN ARCHITECTURE AND IMPLEMENTATION.....	20
9. DEVELOPMENT METHOD	21
9.1. DEFINITION OF THE ARCHITECTURE.....	21
9.2. ALGORITHMS DEVELOPMENT	21
9.3. INTEGRATION	21
10. CONCLUSION.....	22
10.1. FURTHER DEVELOPMENT	22
10.2. RESULTS.....	22
10.3. INTERESTS	22
11. BIBLIOGRAPHY	23

Index of diagrams

Figure 4.1 : Diagram of the system	6
Figure 5.1 : Opening a connection	7
Figure 5.2 : Message reception	8
Figure 5.3 : Message sending	10
Figure 6.1 : Packages hierarchy.....	11
Figure 6.2 : The structure of a message with our Message class	13
Figure 6.3 : Architecture of our project (as in CVS).....	13
Figure 7.1 : The ReAssembly class	17

1. Abstract

A team of France Télécom R&D, the research and development unit of the well-known telecommunication giant France Télécom, is designing a kernel in Java to use it on distributed platforms.

In this project, we were in charge of implementing the protocols stack, which allows the computer to communicate through a network like Internet.

This protocols stack had to be easy to configure and as compact as possible, since it will have to run on different platforms, with maybe memory constraints and different needs.

We have chosen a stack structure with Protocol and Session objects : when you want to add a protocol, you create a Protocol object which will control the Session objects ; the Session objects are used by an application which wants to open a connection with a remote host. If you want to remove a protocol from the stack, you just have to destroy the Protocol object.

After the architecture of the stack has been defined, we have implemented successfully ARP, IP, UDP and ICMP. Since these protocols have some overlaps, some concessions to our architecture have been made.

Our stack has been compiled on the target machine, but since the kernel is not finished, final tests were not possible, but first tests were encouraging.

2. France Télécom R&D

France Télécom R&D, auparavant appelé le CNET (Centre National d'Etude des Télécommunications), est l'unité de recherche et développement de l'opérateur de télécommunications France Télécom, géant présent dans 75 pays et au chiffre d'affaires de 27 milliards d'euros.

France Télécom R&D compte 3800 salariés répartis dans neuf sites en France, dont un à Grenoble. Celui-ci abrite entre autres le département ASR (Architecture de Systèmes Répartis) de la branche DTL (Direction des Techniques Logicielles). La DTL a pour mission d'effectuer des prestations au service des autres directions et de mener des études exploratoires sur les techniques novatrices en matière de traitement de l'information. Le rôle du département ASR est d'étudier et de définir des architectures pour systèmes répartis ouverts, d'organiser une veille technologique, et d'offrir une assistance technique aux autres directions en matière de systèmes distribués.

C'est donc dans le cadre d'un projet de réalisation d'un exo-noyau en Java que nous avons été accueillis au sein du département DTL/ASR pour concevoir la pile réseau nécessaire à ce noyau pour communiquer avec d'autres machines.

3. Aims

The aim of this project is to implement a network stack for the kernel that a France Télécom's team is designing. This kernel is meant for several platforms, with different constraints, that's why our stack must be easy to configure : some protocols would not be useful in some case, so we have to allow the administrator to configure the stack as he wants, and even to add or remove a protocol dynamically. The stack must also be as compact as possible, since the platforms on which the kernel would run could have memory constraints.

For this project, we have to implement at least IP, ARP, ICMP and UDP.

4. Working environment

4.1. The kernel

The France Télécom R&D's DTL/ASR department is designing a distributed objects platform for Power Macintosh machines (using RISC PowerPC processors), allowing the applications to adjust the system to their constraints (real time, quality of services, ...). The objects can communicate together thanks to a μ -ORB, which offers methods call mechanisms and allows to manage links between local and remote objects. The local objects can have access to material resources via an abstraction library: the nano-kernel.

A nano-kernel is not a kernel, it's just a set of drivers which can be viewed as a base for creating some standard kernels (they are then called "personalities"). The nano-kernel allows the personalities to manipulate the processor easily and efficiently. It doesn't provide any process model, any memory manager, etc; yet it must provide interfaces allowing the personalities to construct them easily.

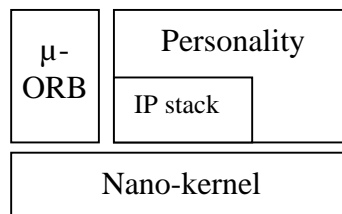


Figure 4.1 : Diagram of the system

In this system, our stack will appear as a library using nano-kernel functionalities in the personality.

As the end of our project, the nano-kernel was on the way of achievement, and so we could use a functional version to test our stack on the target machines. But the personality was not yet developed, and we didn't have services like threads management at our disposal.

4.2. Tools

For this project, we have worked on PC with Pentium III processors, under Linux Debian operating system. The crash machine was an Apple Macintosh G4.

Since our stack has been developed in Java, we used the JDK 1.2 to compile and test our code on Linux computers. For the target machine, France Télécom R&D has designed a native compiler (based on IBM's *Rock* compiler); since this compiler is made for Java 1.1, we had to be careful and not use unimplemented methods or objects (i.e. an object present in JDK 1.2 and not in JDK 1.1).

Thanks to the *javadoc* tool of JDK 1.2, we have constructed easily an HTML documentation of our objects. The only constraint was to comment specially the beginning of our methods, and *javadoc* has generated automatically all the HTML pages.

5. Functional architecture

The functional architecture of our IP stack is inspired from the architecture of x-kernel as described in [Pet 96]. It is based on the utilization of two types of objects, the protocol objects which are responsible for managing the various opened sessions, and the session objects whose role is to transmit data and manipulate the messages (the headers of the messages, actually). It also introduces the notion of “Protocol Graph” that represents the links between the protocols.

5.1. Initialization

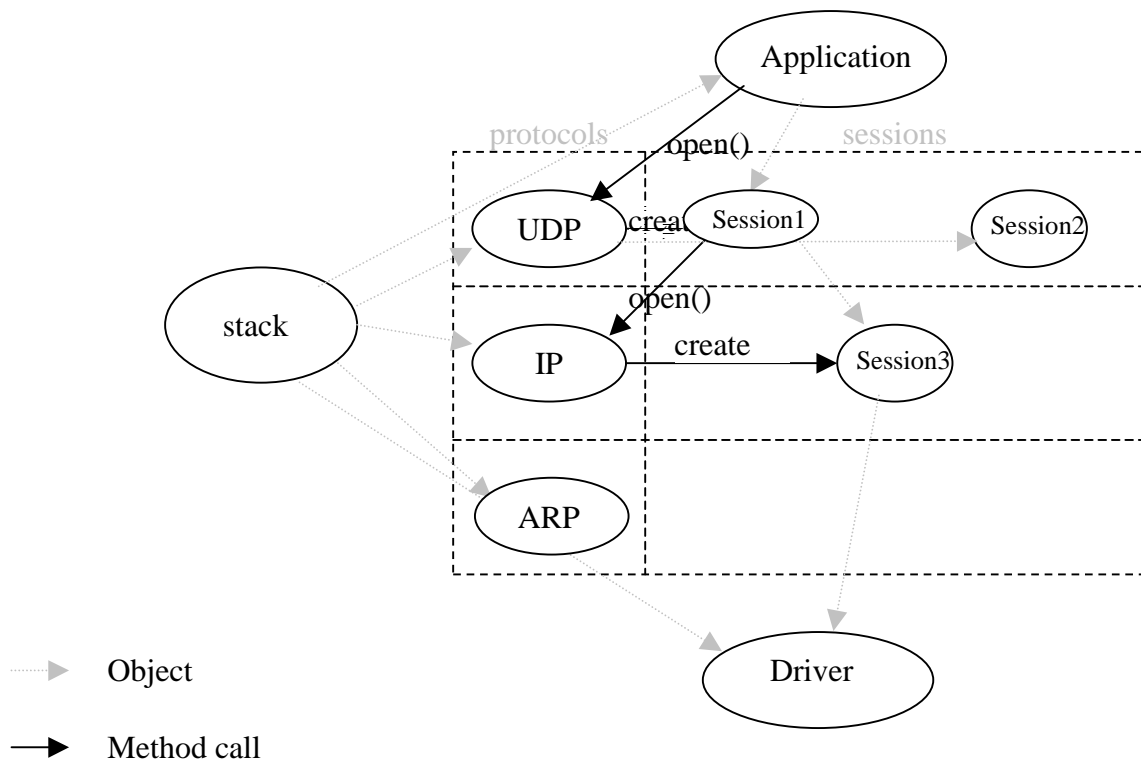


Figure 5.1 : Opening a connection

The *Stack* object is in charge of the initialization of the stack and all the dynamic operations (stack configuration, protocol adding or removing, ...). It also contains the hash table used to store the protocol graph. When the stack is initialized, the *Stack* object creates one Protocol object for each protocol in the protocol graph, and registers it in its hash table. Later, a Protocol object will be able to know what protocols are above or below it by asking the Stack object. Dynamically, we can add or remove a protocol by adding or removing a Protocol object in the hash table.

The Driver implements the *Driver* interface which belongs to the *network.stack.hardend* package and allows the communications between the stack and the hardware.

During the initialization of the stack, the *Stack* object creates one Protocol object for each protocol (one for IP, one for UDP, ...). These Protocol objects are in charge of the initialization and the “destruction” of the Session objects, whose goal is to transmit the messages through the protocol stack to the good host.

A Session object is specific to a connection, and is opened just for it by the Protocol object. Of course, the meaning of the word “connection” depends on the level in which we are: an IP Session will be able to be used by several UDP Sessions since it concerns a pair of hosts; for the IP Session, the connection is between the two hosts, and for an UDP Session between two ports.

Let’s look at a simple example: an UDP connection.

The *Stack* object begins to call the *open* method on the UDP Protocol object: this call creates a new UDP Session object whose reference is registered in the Protocol object and returned to the user interface. This UDP Session calls also the *open* method on the IP Protocol object: if a connection with that remote host has already been established and is still active, the IP Session object exists and can be used, otherwise it is created. The reference to the IP Session object is returned to the UDP Session that now will use it to transmit messages. Considering that in this example IP is the last protocol used in this stack (except the Ethernet protocol), the IP Session can now transmit messages through the network to the remote host.

On its side, the remote host must be waiting for a connection, which means that an application must have called the *openEnable* method on the concerned protocols. This method creates listeners that can receive data from any host. A listener is just a reference in the Protocol object indicating that a remote host can open a connection. When a message arrives for this connection, a Session is created. In our example, there will be one IP listener and one UDP listener (with a fixed port number).

5.2. Incoming messages

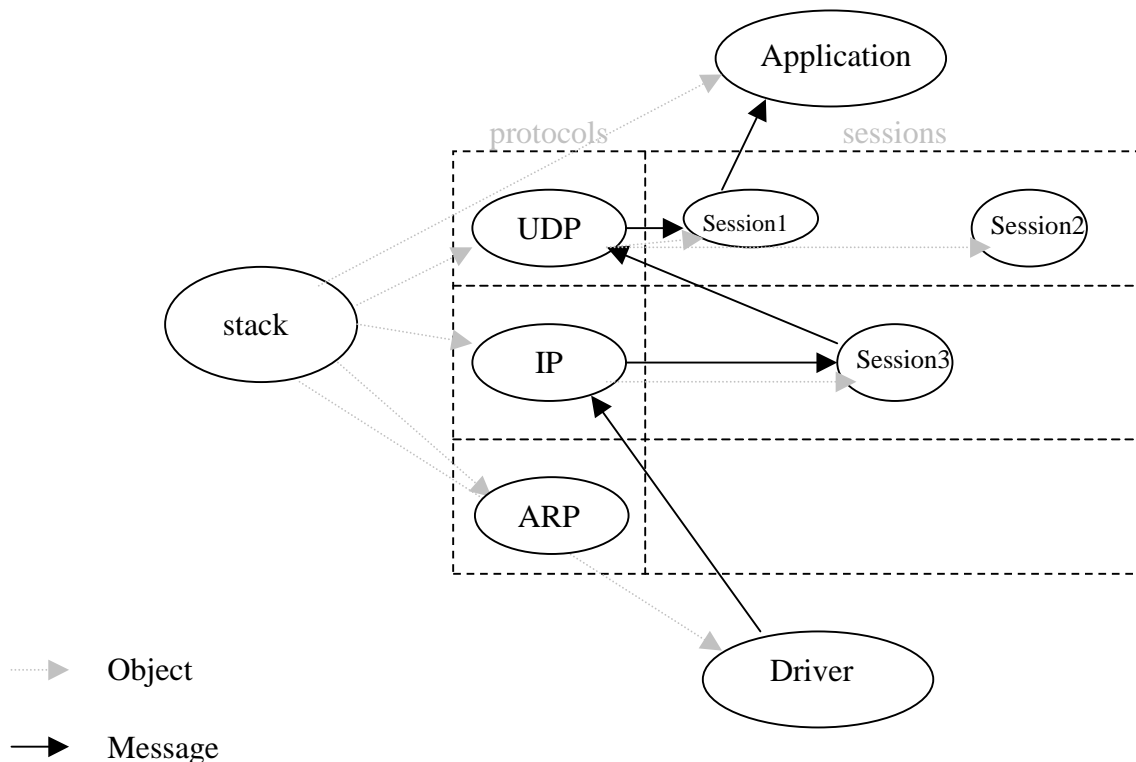


Figure 5.2 : Message reception

When the hardware interface receives a message from a remote host, the driver can determine what protocol is above thanks to the protocol identifier found in the header of the received frame. Then it calls the *demux* method on the Protocol object of this protocol, which cuts the header and transmits the message to the appropriate Session object of its level with a call to the *receive* method. This Session object will call the *demux* method on the higher level Protocol object, and so on, until the end of the stack, when the message is passed to application.

We can use the previous example again: when it receives a message, the hardware interface calls the *demux* method on the IP Protocol objects, which looks at the header to know the sender and the receiver. Then sends this header and the rest of the message to the good IP Session. Thanks to the demux key which is in the IP header, the IP Session knows that the next protocol is UDP. So, it calls *demux* on the UDP Protocol object, which cuts the header and sends the data to the good UDP Session. Then, the UDP Session will transmit the message to the user interface.

5.3. Outgoing message

When you have opened a connection, sending a message to a remote host is very simple, since you just have to go down the stack through the Session objects by calling the *send* method on each of them to add the headers.

In our now well-known example, the application calls the *send* method directly on the good UDP session object, since it knows it after the connection opening. The UDP session adds the UDP header to the message, and calls *send* on the below IP session. The IP session adds the IP headers, and sends the message to the hardware interface which will transmit it to the network.

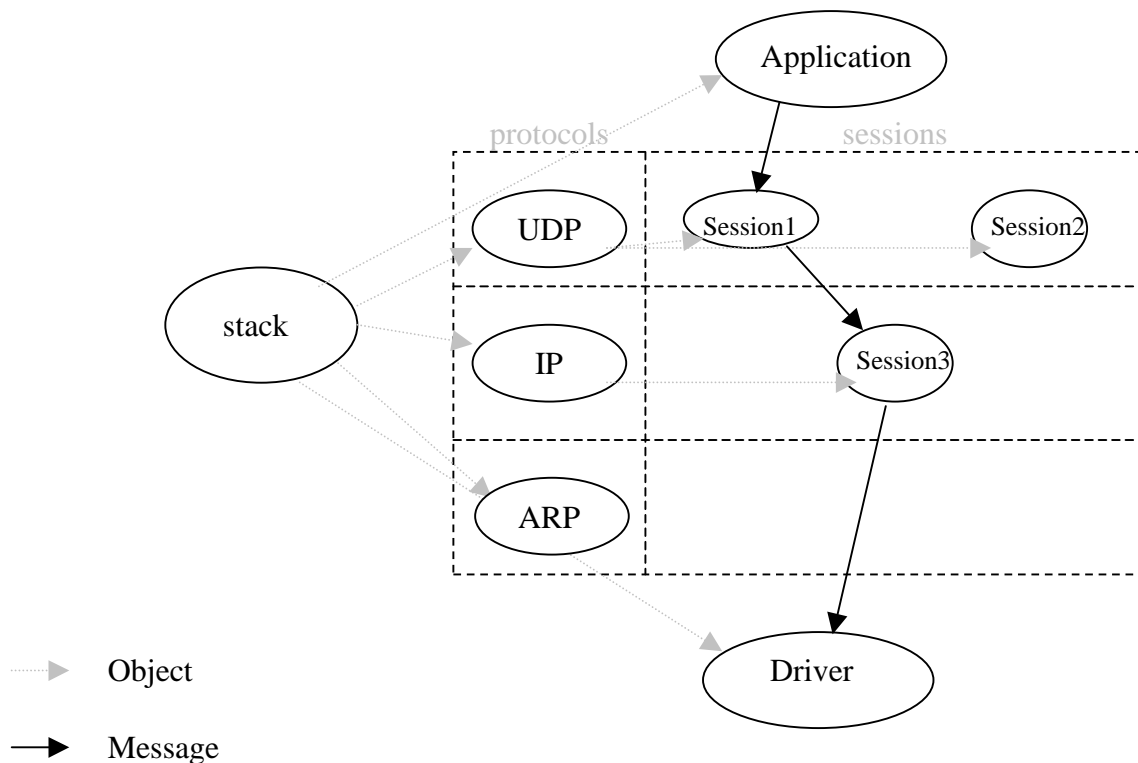


Figure 5.3 : Message sending

6. Software architecture

6.1. Packages hierarchy

To develop this network stack, we need to build a hierarchy of Java packages. In these packages are found the classes that will realize the actual implementation of the network stack. Here is the hierarchy of the packages:

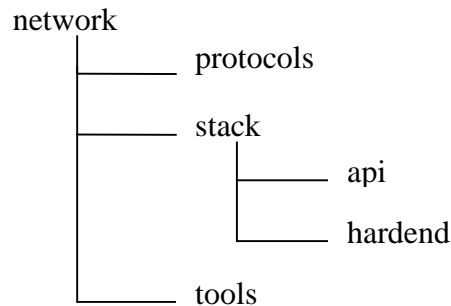


Figure 6.1 : Packages hierarchy

6.1.1. Network.protocols

In this package will be found all the protocols that the network stack will be able to handle. For example IP, ICMP, UDP, etc.

For each protocol you'll find in this package one class that implements the *ProtocolLow* and the *ProtoHigh* interfaces and one other that implements the *SessionLow* and the *SessionHigh* interfaces. Implementing these interfaces is necessary in order for the stack to be able to manipulate these protocols.

The protocol interfaces define the methods necessary for their utilization within the network stack. There are one primary interface called *Protocol* and two others that are extending this one : *ProtocolHigh* and *ProtocolLow*.

ProtocolHigh defines the methods that are related to the opening of a connection or a listener. It must be implemented by a protocol to allow to opening of sessions and listeners. It is also implemented by the *Driver* abstract class that the actual hardware drivers will extend.

ProtocolLow defines the demux method, this method is the one through which a lower-level session can send a message to a protocol. This interface must also be implemented by a *Receiver* in order to receive messages from the stack.

Like the protocols, the sessions have two interfaces.

SessionHigh defines the methods related to the sending of a message and other functions like closing the session, getting the mtu and the total size of headers. This interface is also implemented by the *Driver* to allow it to behave like a session.

SessionLow defines the *Receive* method which is called by the protocol to send the message to the session.

6.1.2. Network.stack

This module is the one in which will be found the classes implementing the core functions of the stack (message and protocol management). This class contains the methods that allow the

management of the stack (add/remove protocols or interfaces) and open connections or listeners. There is one object of this class in the system and it is the one in charge of building the rest of the stack at initialization.

6.1.3. Network.stack.api

In this module can be found the classes or the interfaces related to the user application. For example, the interface *Receiver* must be implemented by an application to be able to receive data from the stack. There is no interface required for the sender.

6.1.4. Network.stack.hardend

On the other end of the stack it is necessary to communicate with the hardware. This package is the one in which will be found Java part of the driver for the network peripherals. This Java part of driver will have to extend the abstract class *Driver* in order to be used by the stack. *Driver* is an abstract class that implements some methods of *ProtocolHigh* and *SessionHigh* interfaces, the others are to be implemented by the developer of the hardware driver.

6.1.5. Network.tools

In this module will be found all the classes used for building the network stack. For example the class that will implement addresses will be found here, like the one implementing the messages. In this module will also be the class used to configure the network stack at compilation time.

The *Message* class is used to represent and manipulate the messages sent and received by the stack. It allows the creation of a message structure by the application or the driver and use it to get the message through the stack to the application.

A message is composed of a chain list of *Message* objects. A *Message* object contains a byte array, an index to the first byte of data, the length of the data, and a handler to the next *Message* object.

Here we point out one limitation of Java: it is impossible to cut one byte array into two without copying the data, like we can do in C or C++ with pointers. Then, to add a header or to divide a message without useless copy, it is necessary to have reserved the space or to have already divided the data into several byte arrays.

When the client wants to send data, he has to create the message as a *Message* objects list according to the MTU of the communication channel and the maximum size of the headers of all the protocols used: the size of the byte arrays contained in the *Message* objects will be equal to the MTU and a space equal to the maximum header size will be reserved at the beginning of these arrays. In the *Message* class can be found a lot of methods to easily handle the *Message* objects.

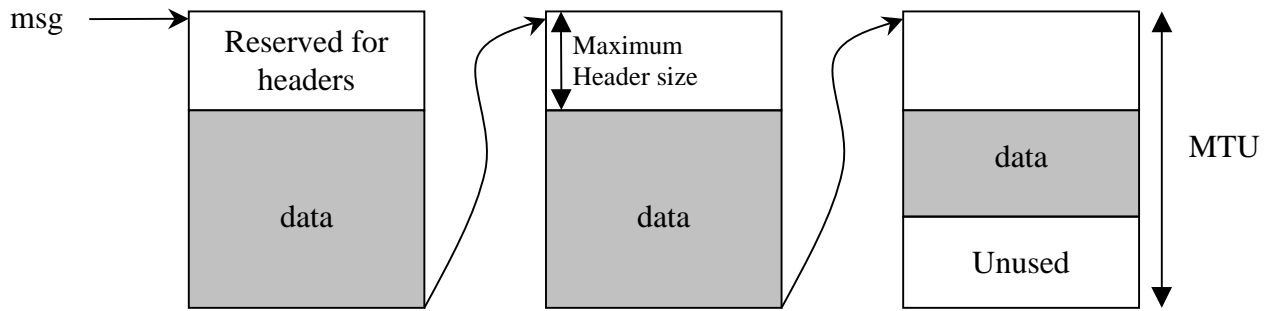


Figure 6.2 : The structure of a message with our Message class

The *Header* class is used to allow the manipulation of the header of a message without copying it. Its structure is nearly the same as *Message*'s, i.e. byte array + index + length of data.

The *Address* classes compose a hierarchy which allows the representation of network addresses. These addresses could be created by the application to inform the stack about the host they want to reach and which resources on their host they want to use (UDP port).

We have defined an empty *Address* interface, to make our architecture as generic as possible. The *IPAddress* class implements this interface and contains only an integer for the IP address. The *UDPAddress* class extends *IPAddress* and adds the UDP port to the IP address. The inheritance allows the client to use an *UDPAddress* object for opening the connection: since an *UDPAddress* is an *IPAddress*, this object will be able to be used by lower protocol than UDP.

6.2. How to

Here are some details about the way to compile our project.

Here is the architecture of our project:

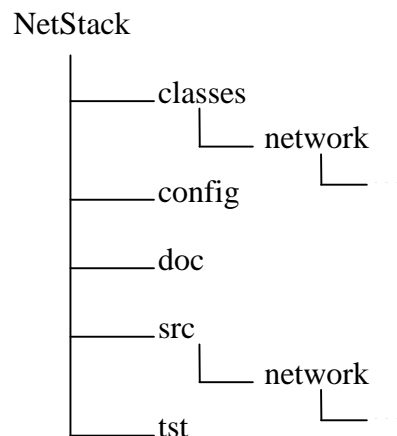


Figure 6.3 : Architecture of our project (as in CVS)

The *classes* directory contains the binaries of our stack, with the same architecture as the packages hierarchy.

The *config* directory contains some tools to help the developer to initialize the stack. To compile correctly the sources, you can precise some compilation variables in the *Make.rules*

file and copy it into the source directories by running *make*. You can also precise the dependencies between the protocols in the *protocols.cfg* file, which is the protocol graph configuration file: if you run “make protograph”, the *ProtocolGraph.java* in the *network.stack* package is written with the modifications, and at the next compilation time changes will be taken into account.

The *ConfigTool* class is a graphic utility to specify values like the debug level or if routing has to be activated or not. Changes are made in the *Config.java* file in the *network.tools* package.

The *doc* directory contains the HTML documentation generated by javadoc.

The source files are placed in the *src* directory, according to the packages hierarchy. In order to compile all the sources, go to the *src* directory and run *make*. If you want to compile only one precise package, run *make* in the corresponding directory.

The *tst* directory contains some tests made during the development of our stack. Some tests will not run because they need temporary methods that we had defined in our classes and that we have removed.

7. Implementation of the protocols

7.1. ARP

ARP is the Address Resolution Protocol: it is the protocol which allows to determine one host's physical address from its IP address. This protocol is needed when using IP over Ethernet. In this case, IP gives the packet to ARP with the IP address of the destination host and ARP is in charge of founding the physical address for this host and sending the packet.

If no physical address can be found for this host, ARP drops the packet. If there are many packets that are sent in a short period of time and ARP does not know (i.e. is waiting for a response) the physical address of the destination host, only the last packet is sent when ARP get the information. This behavior can be justified because usually, the first packet sent between two hosts is a connection request and is shorter than the MTU so there is no need to fragment it and so only one packet to send. In another case, if a big amount of data is sent to one host without prior connection, ARP can't keep all the packets when it wait for an answer. It would take too much memory and may not be useful (if the host is not present on the network).

There is no ARP session since the user doesn't need to use directly ARP. The *ARPProtocol* class contains all what ARP needs to work. Actually, there is only one *ARPProtocol* object in the stack (like the others protocols), but no session; when an IP session need to use Ethernet it access directly methods of *ARPProtocol*. This is one of cases that don't really suit the architecture of our stack. But creating ARP sessions only to make interface between IP sessions and ARP would have cost too much.

7.2. IP

7.2.1. Protocol and Session objects

The *Session* objects are dedicated to the management of an IP connection with a remote host, independently from the high level protocols which are using these sessions (one IP session can be used by several high level sessions).

The *IPSession* objects are referenced by the *IPProtocol* object (one by stack) in a hashtable (called *sessions*). When you call the *openConnection* method, the *IPProtocol* object looks in its hashtable for an existing *IPSession* object corresponding to the remote host; if needed, the session is created and registered in the hashtable. The sessions are stored in the hashtable using the IP address of the remote host as a key, because the client application doesn't need to precise the IP address of the local host it wants to use. Actually, if the local host address is "0.0.0.0", the session checks what interface is used to send the message and gets its IP address to put it in the IP header.

When an IP packet comes from the network, the *IPProtocol* object gets the IP address of the sender of this packet and uses this address as a key to look for a session in the *sessions* hashtable. If a session is found, the packet is sent to it. Else, the *IPProtocol* checks if there is a listener.

The listeners are just references to an high level protocol which can accept connections from any remote host. A listener is created by the *openEnable* method, using the IP address and the high-level Protocol object which are passed as parameters. This IP address is the address under which the listener will be registered. For example, if your computer has two hardware

interfaces and two IP addresses (one by interface), and you want to enable connections only from one interface, you just have to create the listener with the IP address of this interface. If you want to enable connections from all the interfaces, you must give the address “0.0.0.0”. The high-level Protocol object is not used here, so you can just put “null”.

This system is not totally secure, since a remote host can open a session by passing by one interface and send messages to this session by passing by the other interface, so be careful.

To summarize, when the *IPProtocol* receives a packet and there is no *Session* object for that connection, the *listeners* hashtable is glanced through to check if the connection is enabled. If yes, a new *Session* object is created and the packet is sent to it. Otherwise, the packet is rejected.

We have implemented a counter on the *IPSession* objects: this counter is incremented when the *open()* method on the *IPProtocol* object is called, and decremented as the *close()* method on the *IPSession* object is called. When the value of the counter is zero, the *IPSession* object is deleted. If you don't call the *close* method to close a connection, some *IPSession* objects would never be destroyed and the memory could be used improperly.

7.2.2. Fragmentation and reassembly

The fragmentation of a packet is decided in the *IPSession* object, in its *send* method.

If the message has a total length greater than 65515 bytes (65535 minus the 20 header bytes), it is rejected because IP can not manage packets with a size greater than 65535 bytes.

If the MTU of the hardware interface is less than the IP MTU, the message must be fragmented. According to RFC 791, fragments are counted in units of 8 bytes, so we have to break the data portion of a fragmented message on 8 bytes boundaries. So, we have created the *moveAtBeginning* method in the *Message* class: this method moves *n* bytes from the end of the message on which the method was called to the beginning of a given message. We have also reserved 8 bytes at the beginning of every messages thanks to the Maximum Header Size of IP (we have added 8 to this size), knowing that we will have to move between one and seven bytes.

The reassembly is done in the *IPProtocol* object. Why not in the *IPSession* objects? Because the reassembly needs some timeout operators, and for more efficiency we have preferred to concentrate these operations in one object. Actually, when the stack receives one fragmented IP packet, it has to start a timeout on this packet, since if the second packet is lost in the network, the first packet must be destroyed to free memory. Then a thread must be created to check periodically if there is some fragmented packets to destroy. It is more efficient to check in one hashtable in the *IPProtocol* object than in several hashtables in each *IPSession* object, so we store the fragmented packets in the *IPProtocol* object and start a counter. At the creation of the *IPProtocol* object, a thread is started: this thread looks at each entry of the *timeout* hashtable, decrements the counter and destroys packets with a counter equal to zero; then it sleeps ten seconds and does it again. The *timeout* hashtable contains special objects (see the description of the *ReAssembly* class further) that are stored with the identification number and the corresponding *IPSession* as key and that contain a counter and the list of fragments that have already arrived.

Actually, when a fragment comes into the stack, it is sent to the *IPSession* object that sees this is a fragment and sends it to the *IPProtocol* object thanks to the *addFragmentedMessage* that will do the rest: store the packet in the hashtable and start the counter. The counter is

initialized to nine (for a wait of 90 seconds – the RFC 1122 recommends between 60 and 120 seconds) at each time a fragment is added to the message. When the message in the hashtable is complete, it is sent directly to the high-level protocol.

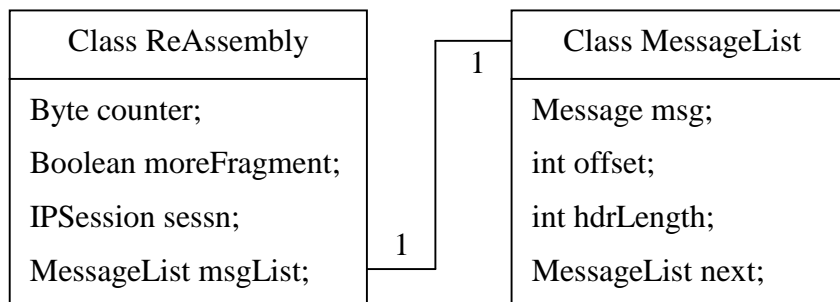


Figure 7.1 : The ReAssembly class

For the needs of the timeout structure, we have defined the *ReAssembly* class in the *IPProtocol.java* file. A *ReAssembly* object contains a counter, a boolean which indicates whether the last fragment (the fragment with the MF flag to false) has been received, a reference to the *IPSession* to which the complete message has to be sent after the reassembly, and a *MessageList* structure in which the fragments are stored.

We have already explained the role of the counter: when its value is zero, the *ReAssembly* object is removed from the *timeout* hashtable. The *moreFragment* boolean is used to know if the last fragment sent by the remote host has been received, and then, if the message is complete: it allows us not to verify it each time we receive a fragment. Until the last fragment is not arrived, we know that the message is not complete. When this fragment is there, we check if the message is complete each time a new fragment has been received. As the message is complete, it is sent to the *sessn* *IPSession* (the IP header of the message was cut, so we have to keep a reference to the session to which the message has to be sent).

The *msgList* attribute is a chained list of *MessageList* objects. A *MessageList* object contains a *Message* object which is one fragment of the original message, the offset of this fragment in the original message, and the size of the IP header. Why the size of the IP header? Because if the counter reaches zero, we have to send an ICMP message to the sender of the message, passing the header of one of these fragments in data. The structure of our *Message* class allows us to get back the IP header using the *reclaimHeader()* method (the header data have not been deleted, the *index* attribute of the *Message* object has just been modified).

7.2.3. Routing

The routing table is stored in the *IPProtocol* object, as a hashtable containing special objects (the class of these objects is internal the *IPProtocol.java* file). If you want to add an entry to this hashtable, you have to precise the destination address (address of a machine or a subnet), eventually the subnet mask, eventually the address of the router, and the hardware interface to use.

If the subnet mask is not given, a value is computed according to the class of the address (A, B or C).

If the router address is *null*, then that means that the host or the subnet can be reach directly.

When a reference needs to be found, the *IPProtocol* object searches if the address is not directly mentioned in the routing table. In other cases, it applies the subnet mask of each entry on the address and compares the result with the subnet address associated to the subnet mask.

The *IPProtocol* object contains also a default entry for the routing table. You can change this entry by giving a null destination address in the *putRoutingEntry* method (assuming you have passed a correct router address). This default entry will be used to send messages whose destination address is not in the routing table.

Since the *Session* objects are dedicated to one precise connection, the routing operations (in case the local host is a router) are done by the Protocol object in the *demux* method. In that case, the local host looks at the destination address, and if the packet is not for it, it sends it to the good host according to the routing table.

7.2.4. Adding a new type of hardware interface

If you want to add a new type of hardware interface (i.e., a new class extending the *Driver* interface), you have to modify some parts of the code of *IPProtocol* and *IPSession*. Actually, the sending procedure depends on the type of the interface. For example, sending a message by Ethernet needs to use ARP, yet by a serial line (PPP) it's not necessary.

Moreover, the IP identifier is not the same for all the hardware-level protocols, and you have to precise it as a parameter in the *send* method. In the Stack object, IP is registered with its identifier for Ethernet (i.e., 2048), but for PPP it's 33, and so on. See RFC 1122 for more details.

So, what are the parts of the code you have to modify? They are two: one in the *demux()* method in *IPProtocol* (for routing) and one in the *send* method in *IPSession* (for sending a message). These two parts are marked by comments.

7.2.5. Type of Services and Options

Our stack architecture is not studied for particular cases like the IP Options. Actually, when we want to send a message, it is not possible to pass other arguments than the message and the identification number of the high-level protocol to a classical *SessionHigh* object through the *send* method. Then, for the *IPSession* class, we have overloaded this method by adding new arguments. There is also two *send* methods :

```
public int send (Message msg, int hlpId)
public int send (Message msg, int hlpId, byte tos, byte[] options)
```

In order to pass a Type of Services byte or an Options bytes array, the client application must use the second version of the *send* method. But to do so, it is necessary to add exactly the same possibility into the high-level protocols like UDP. Then, the *UDPSession* contains also a second *send* method, just for passing arguments to IP.

The *tos* argument is the Type of Services byte, and is directly copied into the IP header. By default (old version of the *send* method), its value is zero. The management of the Type of Services stops here. Actually, if we have had to manage this service, it would have added a high complexity to the routing table. Moreover, the Type of Services is not really used today.

The *options* bytes array must contain the IP options exactly like they should be: this array will be copied at the end of the IP header. Since some options do not have to be copied on

fragmentation but go in first fragment only, they are replaced by the NOP code in the array after the first fragment has been sent.

Of course, the Options are implemented in the *demux* method of the *IPProtocol* class. As a message is received, its options are analyzed and some treatments are done, like recording the route and so on. The only options whose operations were not implemented are the Security and the Stream Identifier options, because they are either badly documented, and the second is totally useless in our case.

7.3. UDP

UDP is a simple transport protocol, it does not include any notion of connection, and only have a checksum system to ensure the data of one packet are well transmitted. It doesn't provide any guarantee of service. But thanks to this it is a simple protocol, easy to implement and that doesn't use lots of resources.

One noticeable thing in UDP is that the checksum that can be found in the UDP header is calculated using also information from IP. This is one of the shortcut in our architecture imposed by UDP specifications.

7.4. ICMP

ICMP is a particular kind of protocol. It doesn't allow data transfer. Its aim is to bring information from one host to another, in particular it is used to inform another hosts about errors that occurred during message processing, or network problems. Because of the aim of this protocol, it has to have a lot of information from other protocols. To do that, we add to the *ICMPProtocol* class some methods to allow the others protocols to call ICMP with all the needed information.

All the error messages are created and sent directly by the *ICMPProtocol* object in order to minimize the memory usage. But ICMP also have a category of messages that are requests and therefore need reply. In order to use this kind of message, we create an *ICMPSession* class that acts like other session classes but needs to be addressed with specific methods. The *send* method does nothing in ICMP; in order to send request, one must use the method specific to each kind of request (*icmpEchoRequest* for example to issue an Echo Request, used for "pinging" another host).

8. Differences between architecture and implementation

The aim of the architecture described in part 5 of this report was to allow a very clean implementation of the protocols. By using the four interfaces (*ProtocolHigh*, *ProtocolLow*, *SessionHigh*, *SessionLow*) one could implement a protocol without having knowledge about the protocol upon or below the one he is developing. Unfortunately, the IP family of protocols has not been designed to work in our architecture, it has been designed about 30 years ago and at the time developers were not very concerned about the “cleanliness” of their implementation. In result, there are a lot of links between the various protocols. In the first implementations, the way a protocol was coded depend on which protocols was to be found above and below it.

Using these protocols in our architecture have lead us to make “shortcuts” between the protocols. In fact to allow the various links between the protocols, we had to add some methods to the protocol or session objects to allow other protocols or sessions to access internal information.

There are many links of this type between IP and ARP since IP uses ARP to obtain the Ethernet address of one host from its IP address. To do that, ARP has been provided with no session, but only a protocol with special method (*ARPResolve*) to allow IP to send packets on an Ethernet network. ARP also needs to access IP information, to know the IP address of an interface when it has to send a request for example.

Between UDP and IP the relation is more simple, the only “shortcut” is that UDP uses fields of IP in its checksum calculation. To allow that, we add two methods, *getLocalAddress* and *getRemoteAddress*, in order to allow the *UDPSession* object to access the two IP addresses of the IP session below it and use them in its checksum calculation.

ICMP is different. This case, there is no need for other protocols to access internal information or for it to access other protocols information. Since this protocol is used to send error or information messages, it needs special methods to do that (instead of just using the standard *send* method). In order to do that, we add special methods to the *ICMPSession* class, for the messages that need to wait for a reply (Echo, Timestamp, ...) and other methods in the *ICMPProtocol* class to allow sending of messages that don't have reply and therefore don't need session to handle it.

In conclusion, the particular case of IP family of protocols does not really suit our architecture but we have make those protocol has “regular” as we can and then added some “shortcuts”. Despite of this, our architecture is not bad since it allows the abstraction of all the internal relations for the application that uses our stack and one can also create really “clean” protocols above (or in place of) the ones that already exist.

9. Development method

This development has been done in three main parts, First the definition of the architecture we have to implement, then the development of the stack in standard Java using a simulation of the network, and at the end the integration of our code in the actual kernel.

9.1. Definition of the architecture

During this first phase, we read a lot on examples of implementation of network stack in order to have a good view of what have been done before in this domain. We choose to build our architecture on the same principles used in x-kernel as described in [Pet 96].

Then we start to define the various Java interfaces needed for our stack.

9.2. Algorithms development

In a first time, we develop all the algorithms of the stack and the protocols using a simulation of the network. We made the simulation using CORBA object to simulate the Ethernet cards and hub, and we also make a simple serial line simulator. This allow us to develop our Java classes without having to compile our code natively or to have the constraints related to the development inside a kernel.

But this way of working introduce a little inconvenient, we weren't able to compare our stack with another one before putting it into the actual kernel. To insure we were not deviating to much from the original protocols, we add a trace in the simulated hub; this trace dumps all the packets in one file formatted to be readable by *Ethereal*, a network packet analyzer. This allow us to check if *Ethereal* recognize our packets (mostly the headers); considering that *Ethereal* was correctly decoding packets, we had a reference to check the correctness of our packets.

9.3. Integration

Once we had developed all our classes we had to integrate them to the kernel. The delicate points were first to change from a simulated network to a real network device driver, and to fit all the constraints of a kernel being currently developed (not all the features were accessible).

After having solved those two problems, all has gone well except for two little problems in checksums calculation that weren't the same as the Linux stack we were confronting to; and *Ethereal* was not checking that those checksum were good.

10. Conclusion

10.1. Further development

When we look at our stack, we can note that it misses a famous protocol: TCP. We have neither the time, nor the need to implement it, but it will be able to be the next step of the evolution of our stack.

We have also looked at the specifications of IPv6: it appears that the architecture of our stack is still adapted. A priori, the development will not present big difficulties; some hard parts of IPv4 have been corrected. The motivation for this new version of IP was to deal with scaling problems caused by Internet's massive growth: IPv6 provides a 128-bit address space (so a new Ipv6 address class will have to be defined) and more efficient mechanisms of fragmentation and options. Because this is out of the subject of our report, we will not explain all the aspects of IPv6: please refer to corresponding RFCs for more details.

10.2. Results

Even if other tests still need to be made, the first tests have shown that our stack is functional. We have successfully established communications between our crash machine and our PCs using UDP, ICMP and ARP, and in simulation, our packets have been successfully recognized by *Ethereal*, a network packet analyzer.

The choice of Java as the development language can be considered as a good and a bad idea: a good idea because Java allows a high level of abstraction and avoids classical programmatic errors (with pointers, for example); a bad because for more efficiency we need sometimes to manipulate directly data in memory, through no "object" layer (see the problems set by the Message class).

10.3. Interests

This internship has allowed us to understand better the running of a protocols stack and the complexity of routing operations. Of course, we had studied the way protocols and networks work during our studies at the ENSIMAG, but we never reach this point in seeing how the protocols are working.

One project like ours allows the mastering of every aspects of a network. It could be interesting for the ENSIMAG students to develop some parts of the IP stack like the routing management in a light version of IP (without the options and the fragmentation) in order to understand better what makes the protocols work.

Moreover, the use of an object oriented language is beneficial because it allows the understanding of the exchange of the messages and then the interactions between the protocols.

Another interesting point was to organize our time and our work to lead to the realization of a project. In this domain, we have seen how it is difficult to make right choices at the beginning concerning the work to do (architecture of the stack) and how to do it (development method).

11. Bibliography

- [Pet 96] *Computer Networks, a system approach*, by **Larry L. Peterson and Bruce S. Davie**, Morgan Kaufmann Publishers 1996
- [Wri 94] *TCP/IP Illustrated*, by **Gary R. Wright and Richard Steven**, Addison-Wesley 1994