PICA user manual

This document presents and describes PICA, a library designed specifically for the implementation of network protocols, which aims at reducing the production cycle duration for MANET’s communication protocols. Nevertheless, it can also be used to write applications that need Operating system features like thread, socket, timer, etc., avoiding having to port them to different platforms.
It also explains how to install and use PICA on different platforms: Windows 2000, Windows XP, Linux, Windows CE 3.0 and 5.0.

Index

1. Introduction

The PICA library was created by Carlos Calafate [1] in order to provide a multi-platform intuitive API for communication protocols’ designers. The objective was to accelerate the prototyping phase to provide users with a stable solution whose source code can compile directly on distinct platforms.
However, PICA is very flexible, allowing users to choose whether to use PICA’s or platform’s features.
The PICA architecture tries to be efficient in terms of code size and speed, making the differences in performance when compared to a customized solution minimal.
The PICA library was developed in ANSI C language, and is available as dynamic-link library (.dll) file for Windows operating systems, and as a shared-object (.so) file for Linux-based operating systems.
PICA is released under the GNU, General Public Licence **¡Error! No se encuentra el origen de la referencia.**.

This manual is divided in four chapters: in the first there is a brief explication about PICA’s overall architecture; in the second there are instructions to install PICA on each platform for which it is available.
The third chapter explains PICA functionalities: if a function uses a particular data structure or defines a new type it is explained in *function description*.
At the end of this manual, there are a table summarizing all data structures and a list of constants provided by PICA.

2. Terminology

Since the biggest difference between operating systems are between Linux and Windows families, in this document the word “Windows” is to be interpreted as the Windows’ operating system family. In case it is necessary to be more precise, we will state the exact version of Windows being used.

3. Overall architecture

The PICA library is an adaptation layer between the users and the kernel space as shown in figure Figure 3-1 and offers specialized functions that aid the programming activity when creating networking solution.

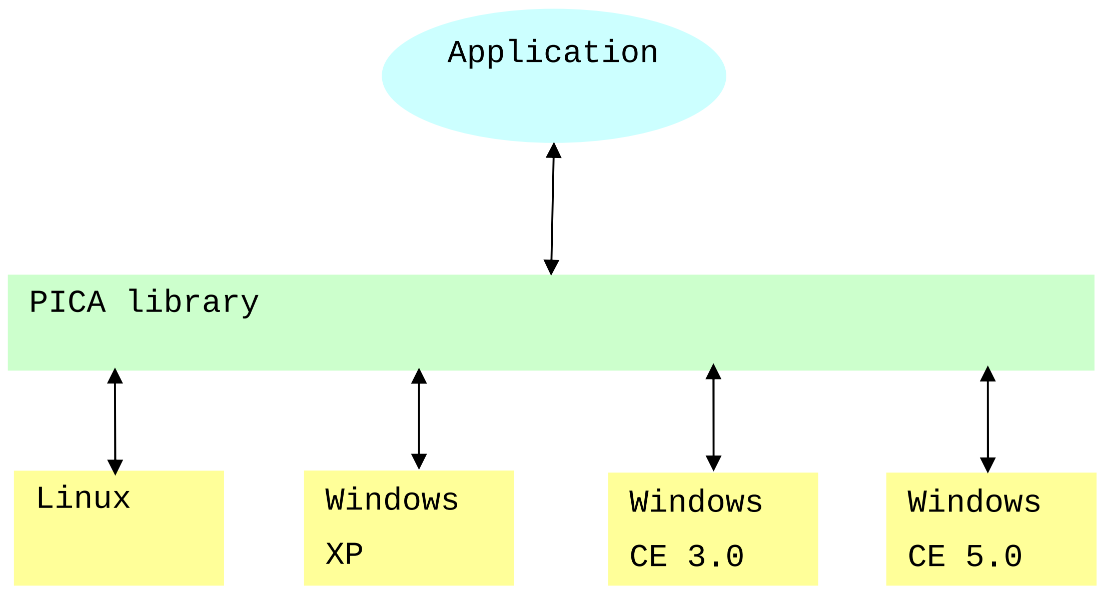


Figure 3-1: PICA architecture

PICA library is based on Winpcap [2], the porting of the Packet Capture Library (Libpcap) **¡Error! No se encuentra el origen de la referencia.**, work by Van Jacobson, to the Windows operating system.
Both Libpcap and its port Winpcap consist of drivers which extend the operating system to provide low-level network access. They also include a library that offers easy access the low-level network layers.
Libpcap allows applications to capture and transmit network packets, bypassing the protocol stack, and has additional useful features, including kernel-level packet filtering, a network statistics engine and support for remote packet capture.

Unfortunately, the port of Winpcap to the Windows CE 5.0 platform is not yet available, but the lack of Winpcap’s port to WinCE 5.0 does not render PICA unusable, since applications can still receive and transmit packets through sockets, but they are mandatory processed by the operating system.
For example, OLSR version 1 and version 2 implementations use PICA without resorting to the Libpcap/Winpcap libraries.

PICA, as shown in following figure, does not completely cover underling layers. Depending on the needs of protocol being designed, programmers can use PICA’s interface, the Libpcap/Winpcap library, or kernel procedures.

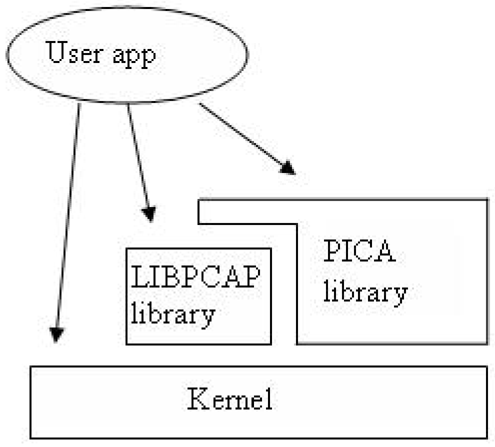


Figura 3-2: PICA architecture

The use of Libpcap library does not compromise inter-platform compatibility, since there is a version of Libpcap for almost all operating systems.
By using kernel specific functions, though, the source code will loose its compatibility between platforms, requiring porting and extra effort on the developer’s side.

4. Installation

This paragraph explains how to create a new application based on PICA, and which external libraries are essential.
PICA is provided with a sample test application showing the PICA procedures used.
The following table resumes necessary PICA libraries and tools used to implement the test application.

	Win2000 - XP	WINCE 3.0	WINCE 5.0	Linux
Library	Winpcap	Packet32	-	-
Tool	Microsoft Visual Studio .Net 2003	ActiveSync 3.5 EmbeddedVisul c++ 3.0	ActiveSync 4.2 Visual Studio .Net 2005	

Tabella 4-1: used tools

Tests on PICA were done with tools' versions shown in the table above; the user can pick a different version, but results are not guaranteed.

The provided PICA package contains these directories:
Linux
Windows
WinCe 3.0
WinCe 5.0

In each directory there is one named "pica" where the PICA library is, and another one named "test" where there is an application for testing PICA. This test consists in one simple print of what happens inside every PICA procedure.

4.1 - *Windows*

The Windows version of PICA is the same for Windows XP and Windows 2000, and it was built with Microsoft Visual Studio .Net 2003.
The library is based on two system libraries: *Iphlpapi* **¡Error! No se encuentra el origen de la referencia.** and *WindowsSocket2* [5] libraries. The first assists network administration of the local computer by enabling applications to retrieve and to modify information about the network configuration on the local computer, while the second allows to work with socket features.

This directory contains two sub-directories:
pica: there are PICA library files (pica.dll and pica.lib) and two directories named “net” and “pica” containing PICA header files.
test: there is Visual studio.Net project test application.

HOW TO USE PICA
In order to install Winpcap download Winpcap executer file from www.winpcap.org/install/default.htm and install it following instructions on that web page.

In order to use pica in a project:
copy the following files and directories into your project’s folder: *pica.dll*, *pica.lib*, *“net”* and *“pica”*;
Add to your project dependencies *“pica.lib ”*;
Insert “#include <PICA/pica.h>” in new code.

4.2 - *WindowsCE 3.0*

PICA’s WinCE 3.0 version was built with Embedded Visual studio C++ 3.0 and it can operate with a WinCE3.0 PocketPC.
It downloads all output files and necessary libraries on PocketPc, but it is necessary to make sure that all of them are downloaded in the same directory (the default one is “*windows**Start*”).

In order to use ip routing functionalities it is necessary to change *“HKEY_LOCAL_MACHINE\Comm\Tcpip\Parms\IpEnableRouter”* registry value from 0 to 1 and perform a soft reset. This change can be done by an utility similar to Windows’s RegEdit, suitable for PocketPc.

WindowsCE 3.0 directory contains two sub-directories:
pica: there are PICA library files (*“pica.dll”* and *“pica.lib”*) and two directories named *“MSInclude”* and *“pica”* containing PICA header files.
test: there is an Embedded Visual Studio project test application.

HOW USE PICA
It is important that your device is connected with your PC through the ActiveSync program in order to allow downloading libraries and applications to the device.

In order to use PICA in a project:
to install necessaries libraries download them from http://www.winpcap.org/install/default.htmt. The download consists in a zip file that contains a project developed with Microsoft Embedded Visual Studio c++. This solution contains three projects named:
DDL
Driver
Sample Apply
While the first two projects consist of code to allow direct interaction with the network interface, the third one is a small application that illustrates the behaviour of the first two libraries.
In order to obtain the Packet32 library you must build the DLL project and download it on the PocketPc; since the Driver project is also required, you must perform the same action on it. Notice that their output files are packet32.dll for DLL and pktdrv.dll for Driver.

Since Winpcap developers do not guarantee its correct functioning, it is advisable to execute SampleApply to verify if the driver is suitable for using with the PocketPC.
when creating a new project enable the socket option. That way the tool initializes the right libraries and the right code for socket use in your project.
Copy the following files and directories, contained in the "ARMDbg" directory of Packet32's project, in the new project's folder: *Packet32.dll*, *packet32.lib*, *PktDrv.dll*, *PktDrv.lib*, "*MSInclude*"
Add the content of PICA's directory to the project's folder.
Add as to the project's dependencies: "PICA.lib".
Insert "#include <PICA/pica.h>" in new code.

4.3 - Windows CE 5.0:

Since Libpcap porting on wince 5.0 is not available, PICA library does not offer functionalities of sending and receiving packets bypassing the protocol stack.
The instructions to install and use PICA are the same ones as for the Wince 3.0 platform, but the tool used is Microsoft Visual Studio 2005, since it allows to operate with Wince5.0 PocketPc, and obviously it is not necessary to add references to the Packet32 library.

4.4 - Linux

In Linux environments, it is necessary to execute all PICA-based applications with root privileges.

This directory contains two sub-directories:
pica source code: includes all the files for PICA;
test: there is a simple project that shows how pica works.
HOW TO USE PICA

In this paragraph you can find a simple instruction to follow for installing and then using pica.

Install PICA:

Run make;

Change to superuser with command:; “su -”

Run “make install”.

The library will be copied to *“/usr/local/lib”* and the headers to *“/usr/include/PICA/”*. Change the *makefile* if you wish to use different paths for the installation.

In order to develop applications make sure to insert “#include <PICA/PICA.h>” in your code

In order to compile your application do: *“gcc <your stuff> -lpica”*

5. PICA library’s internal structure

This paragraph shows the internal structure of the PICA project and PICA’s available features.

The PICA project consists of eleven files of code, and each of them includes a header file.

Figura 5-1 shows the internal structure of Windows’ PICA project and the relationships between header files. Linux’s PICA project has the same internal structure, but without the reference to the Winpcap library.

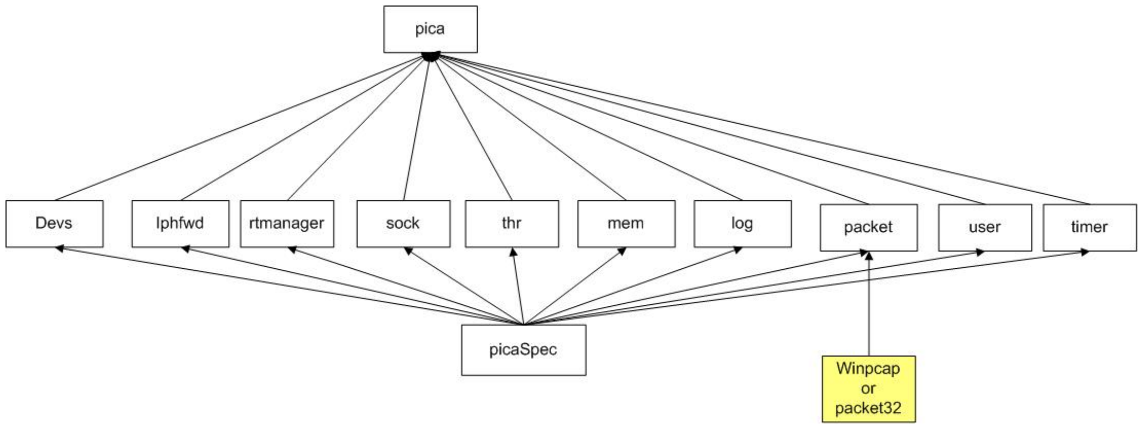
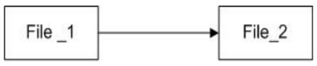


Figura 5-1: PICA internal architecture

In this figure, one blank rectangle is a single PICA header file, while the yellow one is the libpcap or its porting header file.
Arrows are used as follow:



In this case File_2.h includes File_1.h with the directive *#include“File_1.h”.*

6. PICA’s primitives

This paragraph shows and describes PICA’s primitives divided in three logical groups:
process management primitives
memory management primitives
communication management

In each of them, PICA’s primitives are separated by their function.

PICA provides a good error management; each function returns 0 if an error is occurred or 1 otherwise; in the first case it is possible to obtain a description and a code number for the last error that occurred by calling the *PICAgetLastError* function (see paragraph 6.4 - for more details).
Managed errors are almost same in Windows and Linux: some pica functions depend only on system calls, hence the returned error can differ both in message description and in the error number as returned by a function.
Section 6.4 - presents primitives for PICA library management.

Moreover, it is important to notice that, in Windows, the special features, like pipe, mutex and semaphore, used for inter-process communication, are defined “inheritable” in order to allow threads to communicate with main application.

6.1 - System management primitives

6.1.1 In order to write log file

In order to unify file descriptor PICA uses FDesc, that in windows is defined as type *HANDLE*, while in Linux it is defined as type *int*.

```
int PICAopenFile(FDesc * file, char * name, int read_write, int flags)
```

Depending on read_write and flags parameters values, this function opens or creates a file called *name* in read-only mode or read-and-write mode.
The read_write parameter can take the following values:
READF: the file “name” is open in read mode; if it doesn’t exit, the function creates a new one. With this value, the function ignores the flag parameter value.
WRITEF: in this case the function’s behaviour depends on flag value;it can be:
CREATE_CLEAN: Creates a new file in *read_and_write* mode. If the file exists, the function overwrites the file and clears the existing attributes.
APPEND: Open the file called *name* in *read_and_write* mode. If the file does not exist the function creates a new one.

```
int PICAwriteToFile(FDesc file, void * data, unsigned int datasize);
```

This function writes the first *datasize* bytes of information pointed by *data* on the file identified by *file*.

```
int PICAreadFile(FDesc file, void * buf, int buffersize, int * datasize);
```

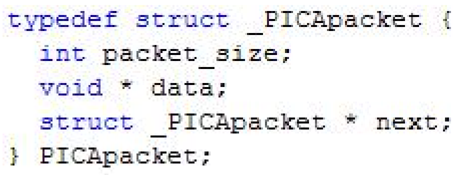
This function read data of maximum length *buffersize* from file identified as “file” and sets *datasize* as the real length of the data read.

```
int PICAcloseFile(FDesc file);
```

This function closes the file identified by “file”.

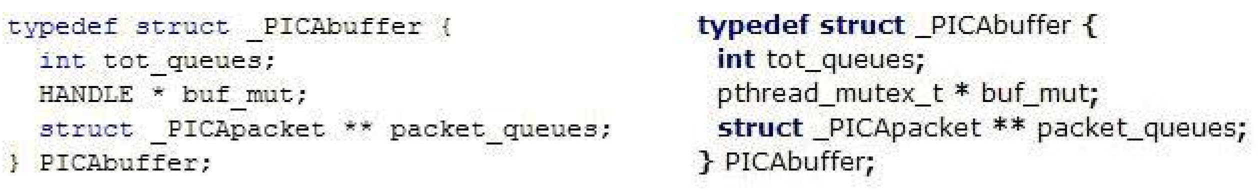
6.1.2 Packet buffer management

Relatively to memory management, PICA’s architecture is based on offering the possibility to easily handle a data structure holding queues. The purpose was to provide auxiliary functions which could be useful for implementation. The user can create as many queues as desired, allowing differentiated packet handling. The architecture chosen allows using multiple groups of queues, each group having a number of queues chosen by the user.
This following figure shows new data structures definitions:



```
typedef struct _PICApacket {
  int packet_size;
  void * data;
  struct _PICApacket * next;
} PICApacket;
```

Figura 6-1: PICApacket structure definition for all platfaorms



```
typedef struct _PICAbuffer {
  int tot_queues;
  HANDLE * buf_mut;
  struct _PICApacket ** packet_queues;
} PICAbuffer;
```

```
typedef struct _PICAbuffer {
  int tot_queues;
  pthread_mutex_t * buf_mut;
  struct _PICApacket ** packet_queues;
} PICAbuffer;
```

Figura 6-2:PICAbuffer structure definition (left: for windows, right: for Linux)

The first represents the information unit to store in the buffer.

Each packet can have a different size.

PICAbuffer is a data structure that contains one or more queues; each of them is a linked list of packets.
In order to guarantee data coherence to multi-threaded applications, PICA's buffer structure has a set of mutexes. Each mutex is used to control the access to each queue so that, for example, distinct threads can read and write to different queues even though these queues belong to the same group.
The mutex set is obtained by using a dynamic array with a number of elements equal to the "tot queues" field. This guarantees data coherence to multi-threaded applications; hence, two or more threads can access different queues in parallel, or to the same queue avoiding the concurrent access problem.
It is relevant to point out that, in order not to generate meaningful delays, these routines do not perform any kind of buffer duplication, having the sole task of managing pointers to data.

```
int PICAinitBuffer(PICAbuffer ** ibuf, int num_queues);
```

This function initializes a PICAbuffer structure with *num_queues* queues and *num_queues* mutex.

```
int PICAaddToBuffer(PICAbuffer * buf, int queue_id, void * data,
int data_size);
```

This function creates a new packet with information pointed by "data" and size of *data_size*. Afterwards, it puts each packet at the end of queue number *queue_id*. It important to notice that the queue number starts at 0.

```
int PICAgetFromBuffer(PICAbuffer * buf, int queue_id, int
num_packets, PICApacket ** packets, int * avail_packets);
```

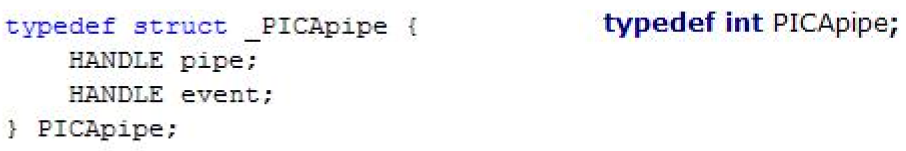
This function gives the first *num_packet* of queue number *queue_id* of buffer *buf*. *Avail_packet* contains the real number of packet retrieved from the queue.

```
int PICAkillBuffer(PICAbuffer * buf);
```

This function frees the memory allocated for the PICAbuffer *buf*.

6.1.3 Pipe management

A pipe is a useful inter-process communication tool. Since pipes are considered as a file under both Linux and Windows, it is represented with a different type: in Linux with the *integer* type, while in Windows with *HANDLE* type.
Moreover, in Windows, the programmer has to signal pipe writing to the reading thread.
Therefore, PICA provides a pipe data structure and primitives to manage it in order to cope with this difference.
Figura 6-3 shows the definitions of PICApipe in Linux and Windows:



```
typedef struct _PICApipe {
    HANDLE pipe;
    HANDLE event;
} PICApipe;
```

```
typedef int PICApipe;
```

Figura 6-3: PICApipe structure definition (left: for Windows, right: for Linux)

Despite Windows¡ pipe declaration uses an event, the communication between the pipe writing thread and pipe reading thread is transparent to the PICA user. (To learn about how to use PICA’s pipe functionalities see the test application.)
On Windows and Linux, reads and writes to anonymous pipes are always blocking. In other words, a read from an empty pipe will block in the call until either one or more bytes arrive, or the pipe is closed and an end-of-file is sent. Likewise, a write to a full pipe will block the call until space becomes available to

store the data being written. Reads may return with less than the number of bytes requested, otherwise known as a short-read.

```
int PICAmakePipe(PICApipe * in, PICApipe * out);
```

This function creates a pipe: the *in* parameter is used to put data, while *out* one to get data on the same pipe.

```
int PICAsendToPipe(PICApipe out, void * data, int size, int * written);
```

This function allows to write data pointed by *data* of size *size* on pipe. The *write* value states how many bytes are actually written.

```
int PICAgetFromPipe(PICApipe in, void * buf, int bufsize, int * datasize);
```

This function allows to read *bufsize* bytes from the pipe and put them in *buf*. The *datasize* value states how many bytes are actually read.

```
int PICAclosePipe(PICApipe pipe);
```

This function frees pipe space.

6.2 - Process management primitives

6.2.1 Timer management:

Windows and Linux represent time in different ways and from different dates: Windows operating system uses intervals of 100-nanosecond intervals since January 1, 1601, while Linux uses a more complex data structure to represent the time expired since January 1, 1970.

In order to unify time representation, PICA represents time values by number milliseconds intervals since January 1, 1970 using UNIT64 type.

```
UINT64 PICAgetCurrTime(void);
```

This function returns current time.

```
int PICAtimer(int action, UINT64 * time, void * function, void * data);
```

This function’s behaviour depends on “action” . Its admitted values are:

T_STARTUP: initializes timer; it is necessary to have done just this action before taking the following actions.

T_SET: sets the “function” and its “data” in the timer’s queue in such a way that all its precedent elements have a timeout value lower that its own, while the next elements have a greater timeout value. Afterwards, it updates the timer to the value of the most recent event. It is important to point out that it is possible to insert two or more elements with a same timer value, and that they will be executed at almost the same time.

T_STOP: removes the timer for the function, therefore deleting the element representing it from the timer queue. The element to remove is identified by both time and function (or by one of them).

T_KILL: terminates the thread created and resets the timer.

6.2.2 Thread management

The fork() call is of common use in Unix environments to manage processes. Windows OS, though, do not offer this function. PICA adopts a combination of the threads approximation with the semaphore and mutex abstractions as an alternative to processes without generating too much extra code. Although the Posix standard doesn’t allow thread suspension and resuming, the PICA

library allows using such functions in the Linux operating system by means of the SIGUSR1 and SIGUSR2 signals.
This solution tries to cope with the differences with respect to the windows kernel where such functions exist. The recommended practice is anyway to avoid such calls because they can produce unpredictable results in critical sections of code. Also, the Posix standard does not allow setting the maximum value of a semaphore, which PICA makes available by introducing a little overhead.

PICASuspendThread and *PICAResumeThread* are primarily designed for use by debuggers. They are not intended to be used for thread synchronization.
Calling *PICASuspendThread* on a thread that owns a synchronization object, such as a mutex or critical section, can lead to a deadlock if the calling thread tries to obtain a synchronization object owned by a suspended thread. To avoid this situation, a thread within an application that is not a debugger should signal the other thread to suspend itself. The target thread must be designed to watch for this signal and respond appropriately.

```
int PICAstartThread(THRID * thr, void * func, void * arg);
```

This function creates a thread identified by *thr* that executes la function *func* with parameters *arg*.

```
int PICAsuspendThread(THRID thr);
```

This function suspends the thread identified by *thr*.

```
int PICAresumeThread(THRID thr);
```

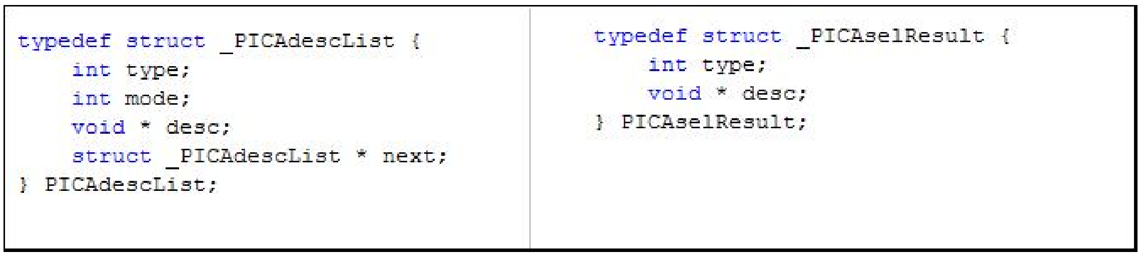
This function resumes the thread identified by *thr*.

```
int PICAkillThread(THRID thr);
```

This function kills the thread identified by *thr*.

```
int PICAselect(int time, PICAdescList * dl, PICAselResult * res);
```

This function emulates the behaviour of Linux's select function. Linux's select function is used to wait for events associated with any kind of descriptor; descriptors are represented by integers values. However, in Windows operating systems, descriptors are generally represented by a specific data type called HANDLE, while integers are only used for sockets. Windows' select function is only available for sockets, while for others events we have to use a function of the *WaitFor* family. The PICA library obviates this problem by emulating the Linux behaviour and using new data structures (*PICAdescList* and *PICAselResult*).



```
typedef struct _PICAdescList {
    int type;
    int mode;
    void * desc;
    struct _PICAdescList * next;
} PICAdescList;
```

```
typedef struct _PICAselResult {
    int type;
    void * desc;
} PICAselResult;
```

These data structures are necessary in Windows because, as referred above, it uses different data types to identify descriptor resources. Therefore, these structures allow identifying type descriptors through the *type* field .

Types admitted are: PICA_PIPE_TYPE, PICA_TIMEOUT_TYPE, PICA_OTHER_TYPE. (see PICAselect function)

PICAdescList represents a list of all descriptors on which to wait for an event.
PICAselResult corresponds to *PICAselect* result; it contains the selected resource descriptor and its type, which is the type of resource it refers to.

```
int PICAaddDesc(PICAdescList ** dl, int type, int mode, void * desc);
```

Add a new file descriptor whose type is *type* and whose mode is *mode*. It is associated with a event described by *desc*.

6.2.3 Mutex management

Windows and Linux/Unix systems manage Mutexes and semaphores in different ways.

Besides having different types for mutex and semaphore descriptors, Linux does not allow setting the maximum value of a semaphore, which is a feature available in Windows. PICA provides this functionality.

In order to cope with the differences PICA provides the following functions to create, operate and destroy mutexes, semaphores and new data structures.

```
int PICAcreateMutex(PICAmutex * mut);
```

```
int PICAcreateSemaphore(PICAsemaphore * p_sem, int initial_count, int max_count);
```

```
int PICAmutexAction(int action, PICAmutex * mut);
```

```
int PICAsemaphoreAction(int action, PICAsemaphore * p_sem, int count);
```

These functions’ behaviour depends on the value of the *action* field:

MUTEX_ACQUIRE or SEMAPHORE_ACQUIRE try to acquire the mutex or semaphore; if it has just been acquired or the semaphore value is

the minimum, the calling thread enters the wait state until the object is signalled or the time-out interval elapses; (blocking the call)
MUTEX_RELEASE or SEMAPHORE _RELEASE: release the mutex or increment the semaphore value.
MUTEX_ACQ_NO_BLOCK or SEMAPHORE _ACQ_NO_BLOCK: to acquire the mutex; if it has just been acquired, the function returns immediately.

```
int PICAdestroyMutex(PICAmutex * mut);
```

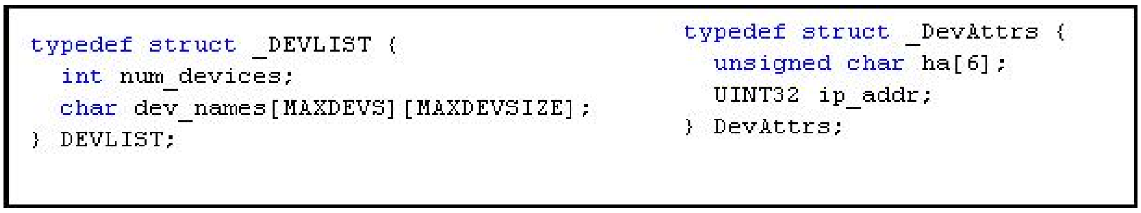
```
int PICAdestroySemaphore(PICAsemaphore * p_sem);
```

6.3 - *The networking management primitives*

6.3.1 In order to get information about available devices

Each device is identified by a different string, depending on the operating system. For Windows operating systems a device is identified by a long and cryptic ASCII string, while in Linux is a string similar to “eth?”, “wifi?”, “ppp?”, etc., where in place of “?” there is a number.
Interesting information about a device are its MAC, net mask and IP addresses.

In order to store this information PICA uses two data structures:



```
typedef struct _DEVLIST {
  int num_devices;
  char dev_names[MAXDEVS][MAXDEVSIZE];
} DEVLIST;
```

```
typedef struct _DevAttrs {
  unsigned char ha[6];
  UINT32 ip_addr;
} DevAttrs;
```

The DEVLIST structure represents a device set containing *num_devices* devices and their identification name. PICA supports a maximum of 128 network adapters on each

DevAttrs structure is used to store the IP and MAC address of a device.

```
int PICAgetAvailableDevices(DEVLIST * devs);
```

This function allows to obtain the number and names of all available devices in *devs*. (see paragraph “PICA data structures” Appendix A)

```
int PICAgetDeviceAttrs(char * dev, DevAttrs * attrs);
```

This function sets in *attr* the MAC, net mask and IP addresses of devices identified by dev. (see paragraph “PICA data structures” pag 14)

6.3.2 Management forwarding information

These functions provide information about forwarding and allow to change its state.

```
int PICAisForwarding(int * true_false);
```

This function gives information about the forwarding status: set *true_false* true if computer forwards packets, false otherwise.

```
int PICAsetForwarding(int on_off);
```

This function sets forwarding on if on_off’s value is FWD_ON, off if _off’s value is FWD_OFF; other values are not admitted.

```
int PICAdefaultTTL(int set_get, int * ttl);
```

This function sets or gets the TTL’s value depending on value of *set_get*:

If its value is TTL_SET then this function sets the TTL with value pointed by *ttl.*
If its value is TTL_GET then this function gets the system's value for the time-to-live.
It should be noticed that the TTL value affected is the global system's TTL. Lowering this value too much might cause loss of connectivity to other networks (e.g. Internet). For a per-socket definition of this value, the appropriate socket option available in most systems should be used.

6.3.3 Frame sending and receiving

```
int PICAopenDevice(char * device, PICA_IO_DEVICE * iodev);
```

This function makes the device identified by name "device" ready to get and put data in the network. After this call, in order to send and receive a packet it is necessary refer to it through the iodev value.

```
int PICAframe(int mode, PICA_IO_DEVICE iodev, void * packet, int packetsize, int * read);
```

This function allows to send and receive packets on device "iodev".
If the "mode" value is:
PICA_SEND: the device sends data pointed by "packet" of size packetsize. In this case parameter "read" is not used.
PICA_RECEIVE: the device receives data from device "iodev" of size "packetsize". "read" value states how many bytes are actually read.
It is important to point out that data to be sent has to conform to the packet format of a MAC frame.
Many network adapters use Ethernet II as packet format. It includes many fields, but the user must create a data stream with only these ones: Destination and source MAC addresses, Ethernet type/length, and Payload, since the remaining ones are set by network adapter.

```
int PICAcreatePacket(char *addr, unsigned char *data, int datasize, unsigned char * packet, int * packetsize);
```

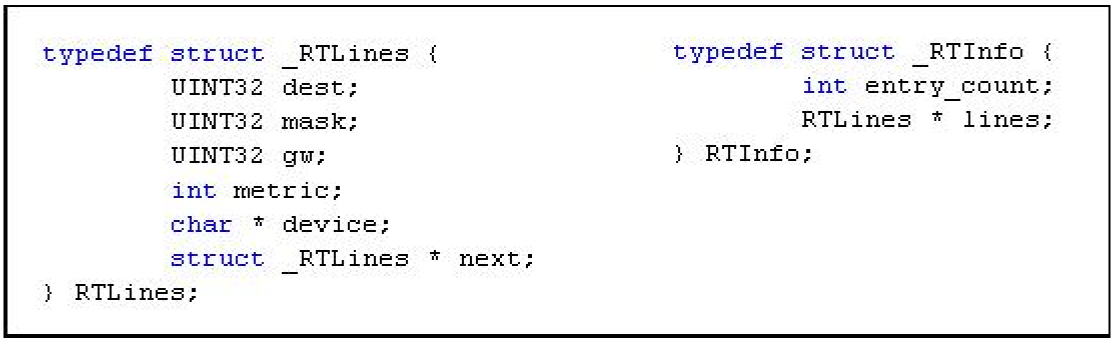
This function creates an Ethernet II packet with the destination address indicated by *addr*, a string with dot-notation and sets the packet data field with the data pointed by *data*.
This function requires data to be already in the hexadecimal format; *datasize* is the length of the data parameter.
Parameter packet will point to the packet just created, and *packetsize* is its size.
This function does not require the source address because the network adapter sets it in each packet before sending.

```
int PICAcloseDevice(PICA_IO_DEVICE iodev);
```

Close the device identified by iodev.

6.3.4 Routing management

These functions allow the user to add and remove entries in the forwarding table and reading such table. They are not frequented used, but we found that they are sometimes useful in protocol design to provide dynamic connectivity. For example in OLSRv1.
In order to transform IP addresses from dotnotation format to bit string it is possible to use … in Windows and … in Linux
The data structures used in Windows and Linux are different. Therefore PICA used the following:



```
typedef struct _RTLines {
        UINT32 dest;
        UINT32 mask;
        UINT32 gw;
        int metric;
        char * device;
        struct _RTLines * next;
} RTLines;
```

```
typedef struct _RTInfo {
        int entry_count;
        RTLines * lines;
} RTInfo;
```

Figura 6-4: RTLines structure and RTInfo structure definitions

RTLines represents an entry in routing table, specifying the adapter to which related.

RTInfo represents routing table with *entry_count* entries number, and lines is a pointer to a list of all entries.

```
int PICAaddRoute(UINT32 dest, UINT32 mask, UINT32 gateway,int metric, char * device);
```

This function adds a route in device's routing table with destination *dest*, mask *mask*, gateway *gateway* and metric *metric*.

```
int PICAdelRoute(UINT32 dest, UINT32 mask, UINT32 gateway, char * device);
```

This function deletes a route from device's route table with destination *dest*, mask *mask*, gateway *gateway* and metric *metric*.

```
int PICAgetRoutingTable(RTInfo * rti);
```

This function gives routing table information stored in the routing table structure (RTInfo) pointed by *rti*.

6.3.5 Socket management

These functions allow the user to open and close a socket.

```
int PICAcreateSocket(PICAsocket * sd, int domain, int type, int protocol, int block);
```

in order to understand the possible values of "domain", "type" and "protocol" parameters see documentation of Windows and Linux socket function [5], while the parameter block can take values:

BLOCK: in order to create a blocking socket, meaning that the application waits for the socket until it receives something.

NO_BLOCK: ehterwise.

```
int PICAcloseSocket(PICAsocket sd);
```

This function closes socket *sd*.

In order to get user information

`int PICAisAdministrator(int * true_false);`

This function sets variable *true_false* to IS_ADM if the process is executing with administrator privileges, or sets it to IS_NOT_ADM otherwise.

6.4 - In order to manage PICA library

`int PICAstartup(int flags);`

This function is essential to use PICA since it initialises PICA library and all necessary data structures. Therefore, it has to be the first PICA function called.

`int PICAcleanup(void);`

This function cleans-up the PICA library, and it must be executed before the application exits. Lack of such call can generate an error.

`int PICAgetLastError(char * err, int * code);`

This is a useful function that allows to debug applications based on PICA: it puts in *err* a concise error description and in *code* its error number code.

7. References

[1] Carlos Calafate www.grc.upv.es/calafate

[2] IEEE, Portable operating system interface (posix) – part 1: System application programming interface (api) [c language]" ISO/IEC 9945-1, 1996.

[3] www.winpcap.org

[4] V. Jacobson, C. Leres and S. McCanne, "The libpcap packet capture library", Lawrence Berkeley Laboratory, Berkely, Ca. Available at http:// www.tcpdump.org.

[5] http://msdn2.microsoft.com/en-us/library/aa366071.aspx

[6] Martin Hall and Dave Treadwel et al., "Windows sockets 2 application programming interface," August 1997, Available at ftp://ftp.microsoft.com/.

8. Index of figures

9. Index of tables