Sistemas Informáticos
Curso 2003-04

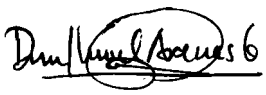
Sniffer_UCM
Herramienta de monitoreo de tráfico

David Manuel Arenas González
María Partearroyo Flores
Mª Montserrat Sanz Sanz

Dirigido por:
Prof. Luis Javier García Villalba
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Los abajo firmantes: David Manuel Arenas González, María Partearroyo Flores y María Montserrat Sanz Sanz, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales, y, mencionando expresamente a sus autores, tanto la presente memoria, como el código, la documentación, y/o el prototipo desarrollado.



David Manuel Arenas González María Partearroyo Flores María Montserrat Sanz Sanz

Índice

1. Objetivo del proyecto

El objetivo del programa es el de espiar tráfico Internet de diferentes máquinas de una red ethernet.

Todo el tráfico espiado tiene que ser mostrado de manera ordenada de forma que el usuario del programa pueda consultar el tráfico de cada máquina independientemente. Además el tráfico captado es mostrado separando los diferentes navegadores que se abren en una máquina y mostrando un historial de navegación de cada explorador web.

Para monitorear tráfico Internet se utiliza la técnica que pone la tarjeta de red en modo *promiscuo*.

La arquitectura del proyecto se divide en tres módulos:

- **Módulo de captura o capturador**, cuya función es la captura de los paquetes que genera el tráfico Internet y que circulan por la red ethrenet.
- **Módulo de enlace de sesiones o enlazador**, cuya función es la de enlazar las distintas sesiones pertenecientes al tráfico de una misma máquina.
- **Módulo de visualización o visualizador**, cuya función principal es la de mostrar al usuario de forma ordenada todo la información capturada.

Target of project

The target of this project is spy Internet traffic of some computers in an ethernet net.

All spy traffic must be show in order. The user can see all traffic of each spy computer separately. Also, the capture traffic is shown like a navigation list of each web explorer.

For sniffing Internet traffic we have used a technique that set the net card in a promiscuous mode.

The architecture is divided in three modules:

- **Capture module**, whose main purpose is to capture packets of Internet traffic in the ethernet net.
- **Session link module**, whose main purpose is link all sessions that belong to a same computer.
- **Show module**, whose main purpose is show to the user all the information captured in order.

2. Manual del programa

El programa se ejecuta en la línea de comandos de la consola. Para arrancar el programa hay que invocar el comando:

sniffer numero_puerto_Internet lista_de_ips_a_espiar

El significado y uso de cada parámetro del programa es:

- **Numero de puerto Internet:** este parámetro hace que se filtren todos los paquetes cuyo puerto TCP fuente o destino sea el número indicado en la línea de comandos. Como el objetivo del programa es capturar tráfico Internet los puertos más comunes son el 80 y el 8080. Se puede poner cualquier tipo de puerto, pero es aconsejable que se ponga siempre este parámetro al valor 80 para la captura del tráfico Internet.

- **Lista de ip´s a espiar:** este segundo parámetro en el fondo son un número indefinido de ellos. Es una lista con un número variable de ip´s separadas por espacio e indican las ip´s de las máquinas las cuales se van a espiar.

Como ejemplo vamos mostrar cual sería el comando para ejecutar el programa que quiere espiar tráfico Internet en las máquinas con ip´s 147.43.54.43 y 123.54.65.65:

sniffer 80 147.43.54.43 123.54.65.65

3.Compilación del programa

El programa utiliza librerías y componentes de libre distribución. Por tanto para ejecutar y poder compilar el programa no es necesario ningún tipo de licencia.

El código está programado para su uso el los sistemas UNIX/LINUX. Se utilizan los lenguajes de programación C y C++ de manera combinada. Solo hay un archivo de código fuente (*sniffer.cpp*).

Se usa la librería STL de C++ por tanto para que el programa compile y pueda ser ejecutado sin ningún problema se tiene que tener instalada esta librería que se puede obtener fácilmente y de manera gratuita por Internet.

Además de la librería STL utilizamos otras dos librerias que tienen que ser referenciadas en el proceso de compilación. Estas librerías son *lpcap* y *lpthread*. Al igual que pasa con la librería STL estas son gratuitas y fáciles de encontrar en Internet.

Para compilar se utiliza el compilador G++. Este compila código C++ conjuntamente con código C. La sentencia de compilación con las opciones más básicas es: *g++ -o nombre_ejecutable sniffer.cpp –lpcap –lpthread*. Donde nombre_ejecutable es el nombre que le queramos poner al archivo ejecutable del programa. Este es el modo de compilación más básico, además existen otras opciones de compilación que están completamente documentadas en el manual de G++.

4. Archivos de registro

El programa según va realizando la captura de paquetes va creando una serie de archivos de registro en los cuales inserta la información más relevante sobre los procesos que se están realizando. Esto facilita la lectura de la información capturada.

Además es un buen sistema de ver como es el funcionamiento del programa y es muy útil de utilizar como archivo de depuración para futuras ampliaciones. Los resultados obtenidos en las capturas pueden ser comparados fácilmente con otro tipo de sniffers u otro programas relacionados, como por ejemplo *ethereal*.

Los archivos de registro que genera el programa con sus características principales son:

- **LogFileTCPSessions:** este archivo muestra en que sesión TCP es tratado cada paquete. Se indica el número de paquete en el orden de captura con las ip´s y puertos fuente y destino. Además se indica si el paquete ha creado una nueva sesión y cual es el identificador de esa sesión o si el paquete es tratado en una sesión ya existente indicando su identificador. Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file tells in what tcpsession is treat each packet *
*                                                          *
************************************************************
------------------------------------------------------------------
Packet 1 --- >with IPSource: 147.96.80.135, IPTarget:
213.4.130.210, PortSource: 32854, PortTarget: 80 create a new
session 213.4.130.210147.96.80.1358032854:
------------------------------------------------------------------
------------------------------------------------------------------
Packet 2 --- >with IPSource: 213.4.130.210, IPTarget:
147.96.80.135, PortSource: 80, PortTarget: 32854 is treat in
session 213.4.130.210147.96.80.1358032854:
------------------------------------------------------------------
------------------------------------------------------------------
Packet 3 --- >with IPSource: 147.96.80.135, IPTarget:
213.4.130.210, PortSource: 32854, PortTarget: 80 is treat in
session 213.4.130.210147.96.80.1358032854:
------------------------------------------------------------------
```

- **LogFileTCP:** este archivo muestra los principales datos TCP de los paquetes capturados. De cada paquete, que se identifica con el número de orden en el que es capturado, se indica cuales son sus ip´s y puertos fuente y destino, el valor del número de secuencia TCP, el número de confirmación (*Ack*), los flags ack, fin, rst y ack y la longitud de los datos HTTP (*dataHTTP*). Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file contains important information of TCP packet   *
*                                                          *
************************************************************
------------------------------------------------------------------
Number packet: 1 -->IPF:147.96.80.135 IPD:213.4.130.210
PortS:32854 PortT:80 Seq:3137642386 Ack:0 fin:0 syn:1 rst:0 ack:0
dataHTTP:0
------------------------------------------------------------------
------------------------------------------------------------------
Number packet: 2 -->IPF:213.4.130.210 IPD:147.96.80.135 PortS:80
PortT:32854 Seq:2497567282 Ack:3137642387 fin:0 syn:1 rst:0 ack:1
dataHTTP:0
------------------------------------------------------------------
------------------------------------------------------------------
Number packet: 3 -->IPF:147.96.80.135 IPD:213.4.130.210
PortS:32854 PortT:80 Seq:3137642387 Ack:2497567283 fin:0 syn:0
rst:0 ack:1 dataHTTP:0
------------------------------------------------------------------
```

- **LogFileHttpPackets:** este archivo muestra todas las peticiones GET y sus correspondientes respuestas en el orden de captura. Tanto las peticiones como las respuestas tienen separadas la cabecera del cuerpo. Para el cuerpo se indica además su longitud si ésta es mostrada en el campo *Content-Length* de la cabecera, sino se mostrará un menos uno. Hay que destacar que tanto una petición como una respuesta puede estar compuesta de varios paquetes. Cuando una petición o una respuesta esta formada por varios paquetes se concatena su contenido adecuadamente y finalmente se inserta su concatenación en este archivo. Este archivo es muy útil para ver de una forma sencilla que todas las peticiones y respuestas se han capturado de forma correcta y que tanto su cabecera como su cuerpo siguen el protocolo HTTP. Además es muy útil para ver el contenido y su tipo en las respuestas. Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file contains the data of a HTTP packet             *
*                                                          *
************************************************************
------------ GET  number:1 ---------------------------------------
---------------
Header Request:
GET / HTTP/1.1
Host: www.terra.es
User-Agent: Mozilla/5.0 (X11; U; Linux i686; es-ES; rv:1.2.1)
Gecko/20030225
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,tex
t/plain;q=0.8,video/xmng,image/png,image/jpeg,image/gif;q=0.2,text
/css,*/*;q=0.1
Accept-Language: es-es, es;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Cookie: GENERAL=1%2C2%2C3%2C4%2C5;
lubid=01013C3C1B600D0019B840501F270000001400000000


Body Request of length -1:
0
Header Response:
HTTP/1.1 200 OK
Age: 15
Date: Tue, 29 Jun 2004 21:05:18 GMT
Content-Type: text/html
Cache-Control: max-age=15
Server: Netscape-Enterprise/4.1


Body Response of length -1:
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=iso-
8859-1">
<base href="http://www.terra.es/">
<LINK REL=StyleSheet HREF=/portada/st2004i.css type=text/css>
//Aqui continua todo el texto html hasta el final
//...
    </script> <script language="JavaScript1.2"
type="text/javascript">var cm8cat="portada";</script>
<SCRIPT language="JavaScript1.2" type="text/javascript"
src="http://adserver.terra.es/multiplescript/checkm8_init_1.js"></
```

```
SCRIPT></body><!--<TRR>21:02:15</TRR>-->
</html>
------------------------------------------------------------------
------------ GET  number:2 ---------------------------------------
Header Request:
GET /ad/es.terra.portada/menu.inmo;tile=4;sz=1x1;ord=81028?
HTTP/1.1
Host: ad.es.doubleclick.net
User-Agent: Mozilla/5.0 (X11; U; Linux i686; es-ES; rv:1.2.1)
Gecko/20030225
Accept-Language: es-es, es;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Accept: video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Referer: http://www.terra.es/
Cookie: id=800000355a1d45d


Body Request of length -1:
0
Header Response:
HTTP/1.0 302 Moved Temporarily
Content-Length: 0
Date: Tue, 29 Jun 2004 19:03:20 GMT
Location:
http://m2.doubleclick.net/viewad/645433/1x1_patrocinio.gif
Cache-Control: private, max-age=0, no-cache


Body Response of length 0:
------------------------------------------------------------------
```

- **logFile:** este archivo muestra las peticiones GET capturadas por el programa en el orden de captura. Indica cual es el número de paquete causante de que la cabecera de la petición GET pase a formar parte de un objeto de la clase *HTTPPacket*. Este caso anterior se da cuando llega el primer paquete de la respuesta a la petición. También se indica cual es el número de paquete causante de que se inserte la respuesta correspondiente en el objeto de la clase *HTTPPacket*. Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file contains GET packets captured by the program *
*                                                          *
************************************************************
------------------------------------------------------------------
Get number: 1
HTTP Header Request:
GET / HTTP/1.1
Host: www.terra.es
User-Agent: Mozilla/5.0 (X11; U; Linux i686; es-ES; rv:1.2.1)
Gecko/20030225
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,tex
t/plain;q=0.8,video/xmng,image/png,image/jpeg,image/gif;q=0.2,text
/css,*/*;q=0.1
Accept-Language: es-es, es;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Cookie: GENERAL=1%2C2%2C3%2C4%2C5;
lubid=01013C3C1B600D0019B840501F270000001400000000

------------------------------------------------------------------
Packet number 5 makes insert complete request
Packet number 6 makes insert response
Packet number 8 makes insert response
Packet number 10 makes insert response
Packet number 12 makes insert response
Packet number 14 makes insert response
Packet number 16 makes insert response
Packet number 18 makes insert response
Packet number 20 makes insert response
Packet number 22 makes insert response
Packet number 24 makes insert response
Packet number 26 makes insert response
Packet number 28 makes insert response
Packet number 30 makes insert response
Packet number 32 makes insert response
Packet number 34 makes insert response
Packet number 36 makes insert response
Packet number 38 makes insert response
Packet number 40 makes insert response
Packet number 42 makes insert response
Packet number 44 makes insert response
Packet number 53 makes insert response
------------------------------------------------------------------
Get number: 2
```

```
HTTP Header Request:
GET /ad/es.terra.portada/menu.inmo;tile=4;sz=1x1;ord=339221?
HTTP/1.1
Host: ad.es.doubleclick.net
User-Agent: Mozilla/5.0 (X11; U; Linux i686; es-ES; rv:1.2.1)
Gecko/20030225
Accept-Language: es-es, es;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Accept: video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Referer: http://www.terra.es/
Cookie: id=800000355a1d45d

-----------------------------------------------------------------
----------------
Packet number 59 makes insert complete request
Packet number 64 makes insert response
-----------------------------------------------------------------
----------------
Get number: 3
HTTP Header Request:
GET /addon/img/elbus/d691b4chenoap.jpg HTTP/1.1
Host: www.terra.es
User-Agent: Mozilla/5.0 (X11; U; Linux i686; es-ES; rv:1.2.1)
Gecko/20030225
Accept-Language: es-es, es;q=0.50
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Accept: video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Referer: http://www.terra.es/
Cookie: GENERAL=1%2C2%2C3%2C4%2C5;
lubid=01013C3C1B600D0019B840501F27000
0001400000000

-----------------------------------------------------------------
Packet number 66 makes insert complete request
Packet number 67 makes insert response
Packet number 69 makes insert response
Packet number 71 makes insert response
Packet number 73 makes insert response
-----------------------------------------------------------------
```

- **logFileCodeType:** este archivo muestra la llegada de paquetes de respuesta de tipo informativo por parte del servidor. Estos paquetes consisten simplemente en una línea de estado y cabeceras opcionales y son terminados en una línea en blanco. Es importante tenerlos en cuenta porque estos paquetes no poseen cuerpo y el servidor manda realmente la respuesta completa pasado un tiempo. Por lo cual simplemente hay que detectar la llegada de paquetes de tipo "1xx" descartando su contenido. En el archivo se muestra el número de paquete en el orden de captura. Una muestra de este tipo de archivo obtenida de una captura real es:

```
****************************************************************
*                                                              *
* This file contains if the response packet is the 1xx type    *
*                                                              *
****************************************************************
```

- **logFileDisconnection:** este archivo de registro muestra una serie de desconexiones numeradas en el orden en que se van produciendo. Para indicar cual es la sesión que se desconecta se indica los puertos del cliente y del servidor. Una muestra de este tipo de archivo obtenida de una captura real es:

```
*********************************************************
*                                                       *
* This file contains information of TCPSessions closed  *
*                                                       *
*********************************************************
---------------------------------------------------------------
Disconnection number 1 of the session 35176 - 80
---------------------------------------------------------------
---------------------------------------------------------------
Disconnection number 2 of the session 35175 - 80
---------------------------------------------------------------
---------------------------------------------------------------
Disconnection number 3 of the session 35174 - 80
---------------------------------------------------------------
```

- **logFileNewSession:** este archivo muestra información sobre las sesiones TCP abiertas. Las sesiones TCP están ordenadas y se muestran en el orden según se van creando. Muestran los datos necesarios para identificar una sesión TCP de manera unívoca, es decir, contiene las direcciones ip y los puertos del cliente y servidor. Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file contains information of TCPSessions opened     *
*                                                          *
************************************************************
------------------------------------------------------------------
New Session: 1
ClientIP: 147.96.80.135 ServerIP 213.4.130.210 ClientPort 35174
ServerPort 80
ClientIP length: 13, ServerIP length: 13, ClientPort length 5,
ServerPort length 2
------------------------------------------------------------------
------------------------------------------------------------------
New Session: 2
ClientIP: 147.96.80.135 ServerIP 209.202.249.250 ClientPort 35175
ServerPort 80
ClientIP length: 13, ServerIP length: 15, ClientPort length 5,
ServerPort length 2
------------------------------------------------------------------
------------------------------------------------------------------
New Session: 3
ClientIP: 147.96.80.135 ServerIP 213.86.246.154 ClientPort 35176
ServerPort 80
ClientIP length: 13, ServerIP length: 14, ClientPort length 5,
ServerPort length 2
```

- **logFileRST:** este archivo muestra los paquetes que tienen el bit de reset a uno. Para identificar que paquete es, se muestra su número con respecto al orden de captura. Una muestra de este tipo de archivo obtenida de una captura real es:

```
************************************************************
*                                                          *
* This file contains TCP packet with the reset flag set    *
*                                                          *
************************************************************
------------------------------------------------------------------
Number packet: 3728 is a reset packet
------------------------------------------------------------------
------------------------------------------------------------------
Number packet: 3729 is a reset packet
------------------------------------------------------------------
------------------------------------------------------------------
Number packet: 3748 is a reset packet
```

- **logFileRepeatGET:** este archivo muestra el número de los paquetes que se intentan insertar en la ventana y no se puede porque es un paquete repetido. Se

muestra su número en el orden de captura. Una muestra de este tipo de archivo obtenida de una captura real es:

```
**********************************************************************
*                                                                    *
* This file contains repeat packets that are not insert into the *
* window                                                             *
*                                                                    *
**********************************************************************
----------------------------------------------------------------------
Number packet: 2592 repeat
----------------------------------------------------------------------
----------------------------------------------------------------------
Number packet: 2593 repeat
----------------------------------------------------------------------
```

5. Módulos de la aplicación

Este es el diseño completo de la aplicación totalmente detallado. Se empieza por una descripción a nivel de modular, para terminar con un detalle a nivel de clases.

5.1 Módulos del programa

Se tienen que lograr tres objetivos principales:

- **Módulo de captura o capturador:** se debe reconstruir la conversación TCP que tiene lugar en la máquina espiada con puerto destino igual a 80. Este es el puerto en el que están escuchando la mayoría de los servidores Web y sobre el que se realizarán las peticiones del cliente.

- **Módulo de enlace de sesiones o enlazador:** hay que capturar los paquetes http intercambiados en la navegación y ser capaz de enlazarlos con las sesiones de navegación.

- **Módulo de visualización o visualizador:** una vez identificadas todas las sesiones de navegación, se deben mostrar en una pantalla auxiliar para su correcto seguimiento.

5.2 Módulo de captura o capturador

Los paquetes capturados por Pcap seguirán el siguiente formato:

Ethernet	IP	TCP	HTTP

En el protocolo IP puede existir fragmentación. Al ser muy poco probable su existencia, para la siguiente explicación asumiremos que no se produce fragmentación IP
Para un tratamiento mas cómodo se crean las clases **IPPacket**, **TCPPacket** y **HTTPPacket**. Las dos primeras inicializan todos sus atributos con el paquete capturado por pcap, sin embargo, un paquete http suele viene “dividido” en varios paquetes TCP debido a la limitación de tamaño de los paquetes que circulan por la red. Para ser capaz de reconstruir paquetes http de petición y respuesta a los mismos hay que realizar una serie de trabajos más tediosos.

Primeramente, cuando un cliente realiza una petición de un recurso Web, se necesita que se haya creado entre su host y el del servidor un canal de comunicación reservando un puerto de su máquina para dicho fin. La clase que se ocupa de controlar y simular ese canal es la **TCPSession** y queda identificada unívocamente por la IP del host origen (el ordenador espiado), la del destino, y ambos puertos.
Cada vez que se captura un paquete se averigua a que sesión TCP pertenece consultando sus IP's y sus puertos. Una vez ligado el paquete con su sesión TCP es necesario reconstruir el flujo TCP entre la máquina origen y destino y simular su ventana. La clase **Window** es la que asume esta tarea. Como la comunicación es bidireccional, por cada sesión TCP existirán dos objetos Window para simular la comunicación en ambos sentidos. Los paquetes enviados en un sentido confirmarán datos recibidos del sentido opuesto y ocasionarán la extracción de bytes de la ventana correspondiente. Estos bytes son parte de la petición http si se obtienen de la ventana *ClientServerWindow* o parte de la respuesta si por el contrario se obtienen de la ventana *ServerClientWindow*. Como los paquetes http vienen "divididos" en varios paquetes TCP, es necesario la creación de clases auxiliares para su reconstrucción. Para esta tarea se han creado las clases **TextHandler, HTTPSession y HTTPPacket.**
TextHandler va almacenando ordenadamente los bytes que constituyen una petición o respuesta http, divididos en cabecera y cuerpo. Esta clase no controla cuándo llega un paquete completo, sino únicamente facilita su almacenamiento y las funciones de parseo.
La clase HTTPSession controla una sesión http, es decir, una petición http por parte del cliente y su correspondiente respuesta por parte del servidor. En una sesión TCP se llevan a cabo varias sesiones http pero siempre en secuencia, es decir, un cliente nunca realizará una segunda petición hasta que no haya recibido la respuesta completa a su petición anterior, por lo que un único objeto HTTPSession por cada TCPSession será requerido. Cada vez que se termina un ciclo, es decir, cada vez que el servidor responde completamente a una petición, se crea un objeto HTTPPacket con la petición y la respuesta que ya pasará a ser enlazado correctamente por el siguiente módulo del programa.

5.3 Módulo de enlace de sesiones o enlazador

Para ser capaz de simular la navegación realizada por el cliente espiado, además de capturar todos los pares petición-respuesta, se deben enlazar los paquetes unos con otros para mantener una secuencialidad temporal y lógica de navegación. En la medida de lo posible se deben identificar el mayor número posible de ventanas de navegación y enlazar las peticiones con la ventana concreta desde la que se hicieron.

Para alcanzar este fin, se han creado las clases **Event**, **EventHandler** y **NavigationSession**.

Cada vez que un cliente realiza una petición mediante la apertura de una página web concreta o pinchando un enlace en alguna ya abierta genera un evento. Un evento es el acto que supone la visualización de una nueva página Web en el navegador, y puede ser ocasionado directamente por el usuario, navegando, o indirectamente mediante pop ups de publicidad, por ejemplo. Cada evento genera una secuencia de peticiones adicionales para la completa visualización de la página que generó el evento. Por tanto, para la carga de una nuevo evento en el navegador del cliente, éste no solamente realiza la petición de la página html principal, sino que solicita una serie de paquetes http adicionales para la carga de imágenes, logos, y demás objetos requeridos.

Un mismo usuario genera multitud de eventos en su navegación por lo que se ha creado un contenedor de eventos, la clase EventHandler**.** Estos a su vez pueden haber sido ocasionados desde uno o varios navegadores Web. Todos los eventos de un mismo navegador se almacenan en la clase NavigationSession.

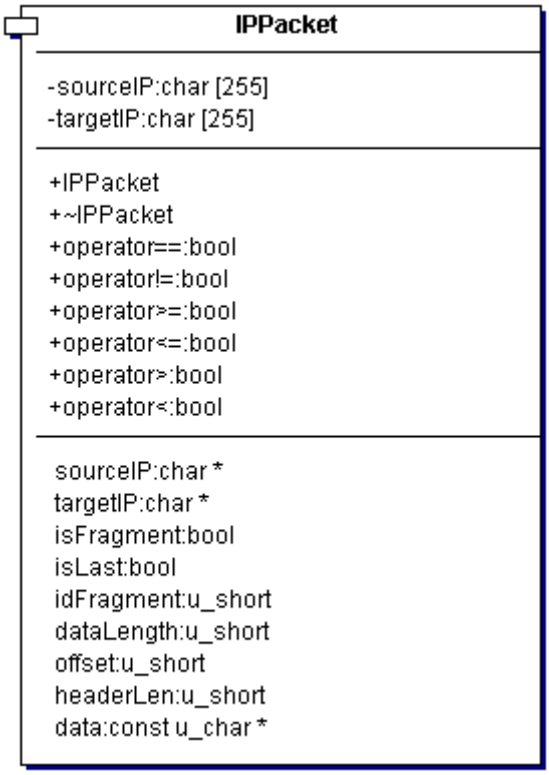
6.Diseño de la aplicación

6.1 Clase IPpacket

- **Funcionamiento general y objetivos de la clase.**

Esta clase guarda todos los datos de la capa IP necesarios para el procesamiento posterior. A partir de esta clase hay que generar la clase TCPPacket. Los datos almacenados se intenta que tengan tipos lo más simples y fáciles de tratar.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **sourceIP**: este atributo almacena la dirección IP fuente del paquete.
  - **targetIP:** este atributo almacena la dirección IP destino del paquete.
  - **data:** este atributo contiene todos los datos del paquete IP es decir la información de todas las capas inferiores a IP (TCP y en algunos casos HTTP).
  - **isFragment:** este atributo es un booleano que es true si el paquete IP está fragmentado y false en caso contrario. Este campo no se puede obtener

directamente de las estructuras de datos ya que es un campo que no viene explícitamente en IP. Para saber si un paquete esta fragmentado hay que examinar los campos *MF* (more fragments) y *offset* (desplazamiento del fragmento). Un paquete no esta fragmentado si MF es 0 y offset es cero, en otro caso el paquete esta fragmentado.

- **isLast:** este atributo solo tiene sentido si el paquete IP esta fragmentado a nivel IP. Cuando hay fragmentación isLast indica que es el último paquete de la fragmentación IP.
- **idFragment:** este atributo solo tiene sentido si el paquete IP esta fragmentado a nivel IP. Indica cual es el número de secuencia del paquete que permite reconocer diferentes fragmentos de un datagrama, ya que todos ellos comparten el mismo número.
- **dataLength:** este atributo contiene la longitud de los datos que contiene el paquete IP.
- **offset:** este atributo solo tiene sentido si el paquete IP esta fragmentado a nivel IP. Indica el desplazamiento del fragmento con respecto al datagrama original empezando por cero.
- **headerLen:** este atributo contiene el tamaño de la cabecera IP.

- **IPPacket (const u_char* packet):** es el constructor de la clase que toma como parámetro el *const u_char** que se ha tomado con libpcap. Para extraer los datos utilizamos las estructuras que nos proporciona el archivo *netinet/ip.h.* Además hay que utilizar funciones auxiliares como *inet_toa (struct in_addr)* que pasa una estructura *in_addr* a un char* en la notación de cifras y puntos y *ntohs(short s)* que transforma el tipo short en la ordenación de la red a la ordenación de la máquina. Para obtener campos concretos es necesario en algunos casos aplicar máscaras como *IP_DF*, *IP_MF*, *IP_OFFMASK*.

```
ethernet = (struct sniff_ethernet*)(packet);
myip = (struct ip*)(packet + size_ethernet);

sprintf(sourceIP,"%s",inet_ntoa(myip->ip_src));
sprintf(targetIP,"%s",inet_ntoa(myip->ip_dst));
idFragment = ntohs(myip->ip_id);
headerLen = ntohs(myip->ip_hl);
dataLength = ntohs(myip->ip_len)-(((u_short)myip->ip_hl)*4);
u_short df = ntohs(myip->ip_off) & IP_DF;
isLast = ntohs(myip->ip_off) & IP_MF;
offset = ntohs(myip->ip_off) & IP_OFFMASK;

dataAux = (const u_char*)(packet + size_ethernet + size_ip);
data=(const u_char*)malloc(dataLength);
```

```
memcpy((u_char*)data,dataAux,dataLength);
```

- **~IPPacket():** destructor de la clase IPPacket que libera toda la memoria utlizada.
- **bool operator XX (IPPacket* p):** estos métodos sobrecargan los operadores ==, <=, >=, <, > y != (se sustituye por XX) para cuando se realicen ordenaciones o comparaciones de objetos de la clase. Toman como atributo de ordenación el *offset* con el orden natural.

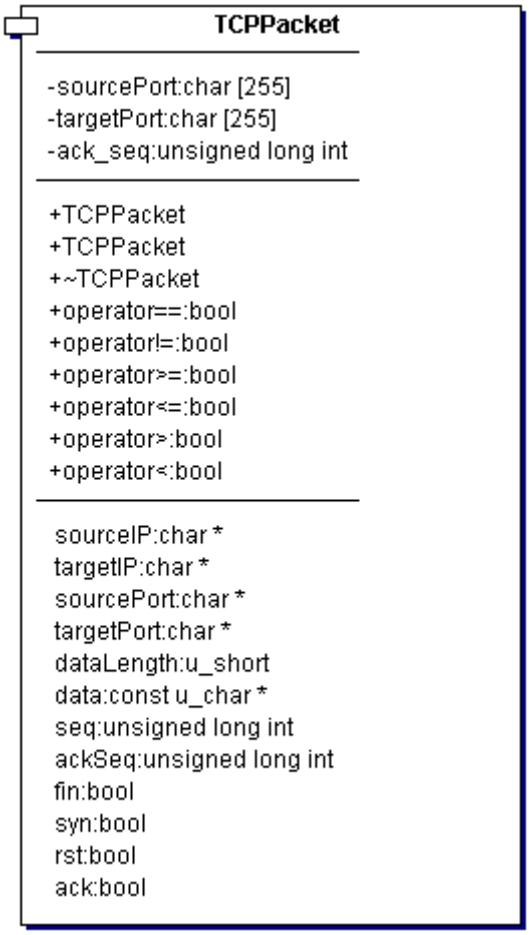
```
bool operator == ( IPPacket p){return offset==p.offset;};
bool operator != ( IPPacket p){return offset!=p.offset;};
bool operator >= ( IPPacket p){return offset>=p.offset;};
bool operator <= ( IPPacket p){return offset<=p.offset;};
bool operator >  ( IPPacket p){return offset> p.offset;};
bool operator <  ( IPPacket p){return offset< p.offset;};
```

6.2 Clase TCPPacket

- **Funcionamiento general y objetivos de la clase.**

Esta clase guarda todos los datos de la capa TCP necesarios para el procesamiento posterior. Los datos almacenados se intentan que tengan unos tipos lo más simples y fáciles de tratar.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **sourcePort:** este atributo almacena el número de puerto fuente que usa el paquete TCP.
  - **targetPort:** este atributo almacena el número de puerto destino que usa el paquete TCP.
  - **sourceIP:** este atributo almacena la dirección IP fuente del paquete TCP.

- **targetIP:** este atributo almacena la dirección IP destino del paquete TCP.
- **seq:** este atributo almacena el número de secuencia del paquete TCP. Este es útil para la reconstrucción de los fragmentos TCP siguiendo el mecanismo de ventana deslizante.
- **ack_seq:** este atributo almacena el número de confirmación o *ack*. Este campo solo tiene sentido si el bit *ack* es 1, es decir, si el campo *ack* de la clase TCPPacket es true. Este número indica hasta que paquete está confirmada la comunicación TCP.
- **fin:** este booleano si es true indica que el emisor no tiene más datos que emitir y es false en caso contrario.
- **syn:** este booleano si es true indica que el emisor y receptor están sincronizando los números de secuencia y es false en caso contrario.
- **rst:** este booleano si es true indica que la conexión se termina de forma abrupta por alguna causa y es false en caso contrario.
- **ack:** este booleano si es true indica que el campo ack_seq tiene sentido y es false en caso contrario.
- **data:** este atributo almacena los datos TCP que en caso de haberlos siempre son datos HTTP.
- **dataLength:** este atributo almacena la longitud de los datos TCP, sin incluir los datos de la cabecera, es decir la longitud de los datos HTTP.

- **TCPPacket (IPPacket* ipPaq):** constructor de la clase a partir de un objeto de la clase IPPacket. Este constructor obtiene todos los datos para los atributos anteriormente citados a partir del atributo *data* de la clase IPPacket ya que los datos de IP son el paquete TCP.

```
mytcp = (struct tcphdr*)ipPacket->getData();

char* aux = ipPacket->getSourceIP();
int length = strlen(aux);
sourceIP = (char*)malloc(length+1);
bcopy(aux,sourceIP,length);
sourceIP[length] = '\0';

aux = ipPacket->getTargetIP();
length = strlen(aux);
targetIP = (char*)malloc(length+1);
bcopy(aux,targetIP,length);
targetIP[length] = '\0';

sprintf(sourcePort,"%i",ntohs(mytcp->source));
sprintf(targetPort,"%i",ntohs(mytcp->dest));
seq = ntohl(mytcp->seq);
```

```
ack_seq = ntohl(mytcp->ack_seq);
auxfin = ntohs(mytcp->fin);
auxsyn = ntohs(mytcp->syn);
auxrst = ntohs(mytcp->rst);
auxack = ntohs(mytcp->ack);

dataLength=ipPacket->getDataLength()-(mytcp->doff*4);
dataAux = (const u_char *)(ipPacket->getData() + mytcp->doff*4);
data=(const u_char*)malloc(dataLength);
memcpy((u_char*)data,dataAux,dataLength);
```

- **TCPPacket (TCPPacket* tcpPaq):** constructor copia de la clase.
- **~TCPPacket():** destructor de la clase que libera toda la memoria utilizada.
- **void setDataLength(u_short d):** este método fija la longitud de los datos a d. Es utilizado principalmente cuando se reconstruyen los fragmentos IP para formar un paquete TCP.
- **void setData (const u_char* c):** este método fija los datos TCP a c. Al igual que el anterior se utiliza principalmente en la reconstrucción de los fragmentos IP para formar un paquete TCP.
- **bool operator XX (TCPPacket* p):** estos métodos sobrecargan los operadores ==, <=, >=, <, > y != (se sustituye por XX) para cuando se realicen ordenaciones o comparaciones de objetos de la clase. Toman como atributo de ordenación el número de secuencia (seq) con el orden natural.

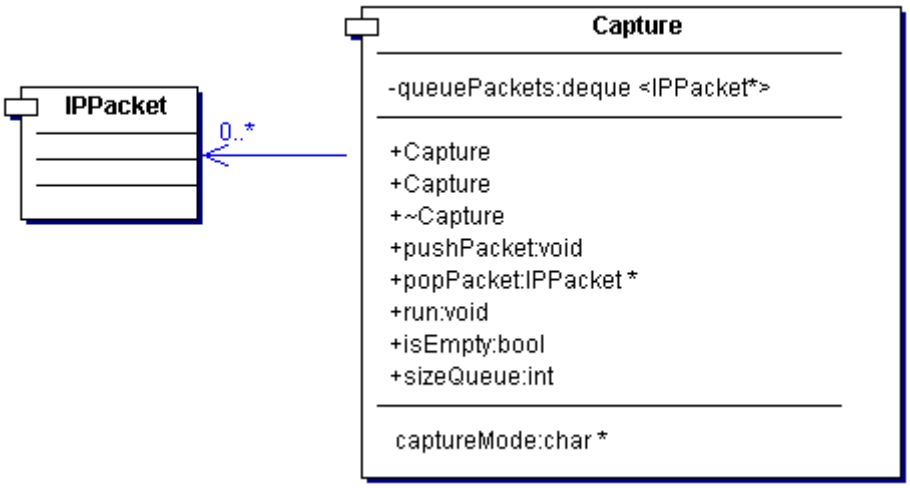
```
bool operator ==( TCPPacket p){return seq==p.seq;};
bool operator !=( TCPPacket p){return seq!=p.seq;};
bool operator >=( TCPPacket p){return seq>=p.seq;};
bool operator <=( TCPPacket p){return seq<=p.seq;};
bool operator > ( TCPPacket p){return seq> p.seq;};
bool operator < ( TCPPacket p){return seq< p.seq;};
```

6.3 Clase Capture

- **Funcionamiento general y objetivos de la clase.**

El objetivo de esta clase es el de capturar todos los paquetes de la red Ethernet que se ajusten a un filtro predeterminado. El filtro hace que solo se capten los paquetes en los que intervienen una lista de máquinas y sea tráfico Internet (HTTP puerto 80, 8080, etc...). La clase captura los paquetes y los inserta en una cola a partir de la cual se extraerán para hacerles el tratamiento correspondiente. La principal función de esta clase es hacer que ningún paquete se pierda e insertarlos todos en un cola en el orden de llegada.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **captureMode:** este atributo es el filtro de captura que se aplica a lpcap. Este filtro indica cuáles son las máquinas cuyo tráfico va a ser capturado y que tipo de tráfico capturar. En nuestro caso pueden ser una o varias máquinas y el tráfico capturado es Internet HTTP.
  - **queuePackets:** cola de objetos de la clase IPPacket. En esta cola los objetos están ordenados en el orden en que han sido capturados.

  - **Capture(char* mode):** este es el constructor de la clase que tiene como parámetro el filtro de captura.
  - **Capture():** constructor por defecto de la clase.
  - **~Capture():** destructor de la clase que libera toda la memoria utilizada.

- **void pushPacket(IPPacket* p):** este método inserta un objeto de la clase IPPacket, que se le pasa por parámetro, en la cola.
- **IPPacket* popPacket():** este método saca un objeto IPPacket* de la cola y lo devuelve si esta no es vacía. Si la cola es vacía retorna *NULL* como indicación de que no se ha extraído ningún objeto de la clase.
- **void run():** este es el método que inicia el capturador. Para ello lo que se hace es crear un hilo de mayor prioridad que el hilo del programa principal. Este hilo solo se encarga de tomar los paquetes que captura lpcap y encolarlos en *queuePackets.* Este hilo está siempre activo y tiene como comportamiento por defecto el coger todos los paquetes de la red que se ajusten al filtro sin ningún tope numérico.

```
int error;
pthread_t idHilo;
error = pthread_create (&idHilo, NULL, threadHandler, this);
```

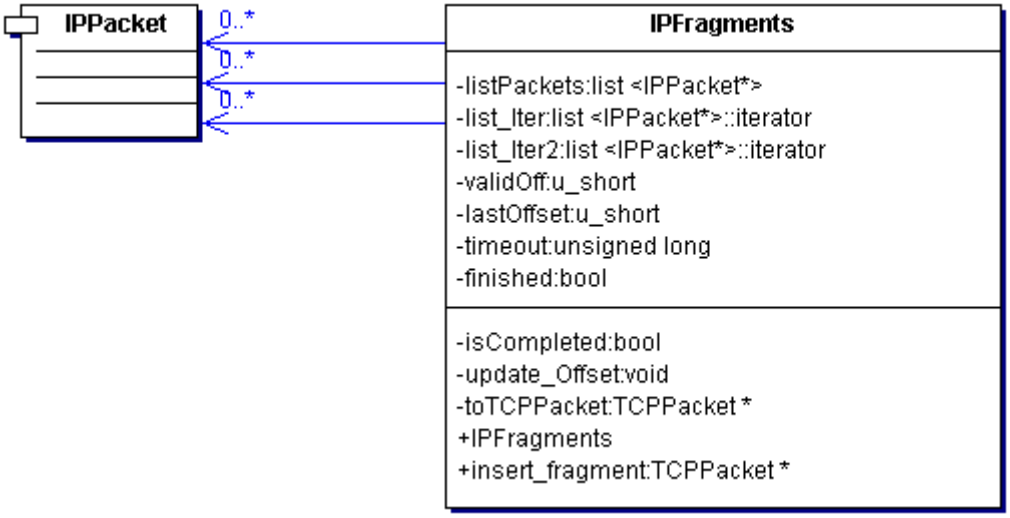
- **bool isEmpty():** este método devuelve un booleano que es true si la cola *queuePackets* es vacía y es false en caso contrario.
- **int sizeQueue():** este método devuelve el número de paquetes que hay en *queuePackets*.

6.4 Clase IPFragments

- **Funcionamiento general y objetivos de la clase.**

El objetivo principal de esta clase es el de reconstruir la fragmentación a nivel IP. Esta clase hace el tratamiento de los paquetes IP fragmentados insertándolos en una cola de prioridad en la cual todos los paquetes están ordenados con respecto a la ordenación dada en la sobrecarga de operadores relacionales, es decir, respecto al *offset*. Se controla que todos los fragmentos han llegado y se hace una reconstrucción formando un paquete TCP. Esta clase es utilizada solo en la clase IPStack que engloba toda la fragmentación IP distinguiendo entre las diferentes fragmentaciones activas.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **listPackets:** cola de prioridad en la cual están los fragmentos que van llegando ordenados por *offset*.
  - **validOff:** este atributo indica con el valor de *offset* hasta que paquete está todo el fragmento completo, es decir, contiene el valor de *offset* del paquete hasta el cual están todos los fragmentos anteriores. Cuando este atributo coincide con el *offset* del último paquete indica que todos los fragmentos han llegado.

- **lastOffset:** este atributo almacena el *offset* del último paquete de la fragmentación IP, si este paquete todavía no ha llegado lastOffset es cero.
- **bool isCompleted():** este método devuelve true si el ya están todos los fragmentos IP necesarios para la reconstrucción del paquete TCP y false en caso contrario.
- **void update_Offset():** este es un método privado el cual tiene como función actualizar el atributo validOff. Para ello va recorriendo toda la cola de prioridad y comprueba que el *offset* del fragmento más su tamaño es igual al *offset* del fragmento siguiente.

```
IPPacket* ipPaq = listPackets.back();
if (ipPaq->getIsLast()) lastOffset = ipPaq->getOffset();

l1 = listPackets.begin();
//l2 contiene la posicion final
l2 = listPackets.begin();
size = listPackets.size()-1;
for(int i=0;i<size;i++){
   l2++;
}

while (l1!=l2 && !fin){
  dlen = (*l1)->getDataLength();
  hdlen = (*l1)->getHeaderLen();

 //l3 se utiliza como auxiliar para acceder al elemento siguiente
  l3 = l1;
  l3++;
  if (validOffAux+hdlen+dlen == (*(l3))->getOffset()){
      validOffAux=(*(l3))->getOffset(
  }
  else fin=true;
  l1++;
}

validOff = validOffAux;
```

- **TCPPacket* toTCPPacket():** este es un método privado cuya función es la reconstrucción del paquete TCP. Esta función solo tiene sentido utilizarla si están todos los fragmentos. Devuelve un objeto de la clase TCPPacket con todos sus atributos completos. Hay dos atributos a los cuales hay que dar un tratamiento especial que son *data* y *dataLength*. *Data* está formado por la concatenación ordenada de todos los datos TCP de los fragmentos y *dataLength* es la longitud de estos datos.

```
IPPacket* ipPaq = listPackets.back();
TCPPacket* tcpPaq = new TCPPacket(ipPaq);
//Al crear el paqueteTCP solo varian los datos y la longitud de
los datos
for (l1=listPackets.begin();l1!=listPackets.end();l1++){
     tcpPaqList = new TCPPacket((*l1));
     dataLength=dataLength+tcpPaqList->getDataLength();
}
data=(u_char*)malloc(dataLength);

for (l1=listPackets.begin();l1!=listPackets.end();l1++){
     tcpPaqList = new TCPPacket((*l1));
     dataAux = (u_char *)(tcpPaqList->getData());
     memcpy((u_char*)data,dataAux,tcpPaqList->getDataLength());
     data=data+tcpPaqList->getDataLength();
}
data=data-dataLength;
tcpPaq->setData((const u_char*)data);
tcpPaq->setDataLength(dataLength);
return tcpPaq;
```

- **IPFragments(IPPacket* ipPacket):** este método es el constructor de la clase que tiene como parámetro un objeto de la clase IPPacket. Al crear el objeto se inserta en la cola de paquetes el paquete IP que se ha pasado como parámetro además de inicializar todos los atributos necesarios.
- **TCPPacket* insert_fragment(IPPacket* ipPacket):** este método inserta un fragmento IP en *listPackets* de forma ordenada según su offset. Además se actualiza la variable validOff. Si cuando se han realizado las operaciones anteriores están todos los fragmentos, se reconstruye el paquete TCP invocando al método privado *toTCPPacket()* y se devuelve el *TCPPacket** creado, si no están todos los fragmentos se devuelve *NULL*.

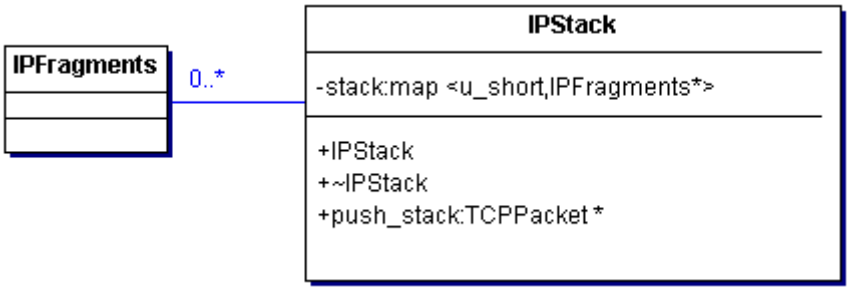
```
if(ipPacket->getIsLast()) lastOffset = ipPacket->getOffset();
listPackets.push_back(ipPacket);

//se ordena la lista
listPackets.sort();
this->update_Offset();
if (isCompleted()==true) return toTCPPacket();
else NULL;
```

6.5 Clase IPStack

- **Funcionamiento general y objetivos de la clase.**

El objetivo de esta clase es controlar toda la fragmentación IP. Por esta clase pasan todos los paquetes capturados. Si no hay fragmentación a nivel IP lo único que se hace es crear el paquete TCP correspondiente al paquete IP. Si el paquete que a va a pasar por la pila IP está fragmentado a nivel IP se busca en una *tabla hash* de objetos IPFragments si existe una entrada para una clave que identifica unívocamente las distintas fragmentaciones. Si la entrada existe, se inserta el paquete en el objeto IPFragments de la entrada, sino, se crea una nueva entrada con un nuevo objeto IPFragments.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **stack:** este atributo es una *tabla hash* que contiene objetos IPFragments. La clave de búsqueda identifica unívocamente las distintas fragmentaciones que están activas, en este caso la clave es el número de secuencia ya que todos los fragmentos IP de un paquete fragmentado tienen el mismo número de secuencia. El objeto que hay en cada entrada de la tabla gestiona la reconstrucción de los fragmentos IP.

  - **IPStack():** este es método constructor de la clase que inicializa la *tabla hash* a vacía.
  - **~IPStack():** destructor de la clase que libera toda la memoria utilizada.
  - **TCPPacket* push_stack(IPPacket* ipPacket):** este método es el que trata toda la fragmentación IP. Toma el paquete que se le pasa por parámetro y se mira si está fragmentado o no para hacerle el tratamiento o pasarlo a paquete TCP directamente. Si el paquete esta fragmentado se busca si hay una entrada en la *tabla hash* correspondiente a el número de

secuencia y si no la hay la crea. Se reconstruye el paquete TCP cuando han llegado todos los fragmentos y se borra la entrada de la *tabla hash* que gestionaba esa fragmentación IP.

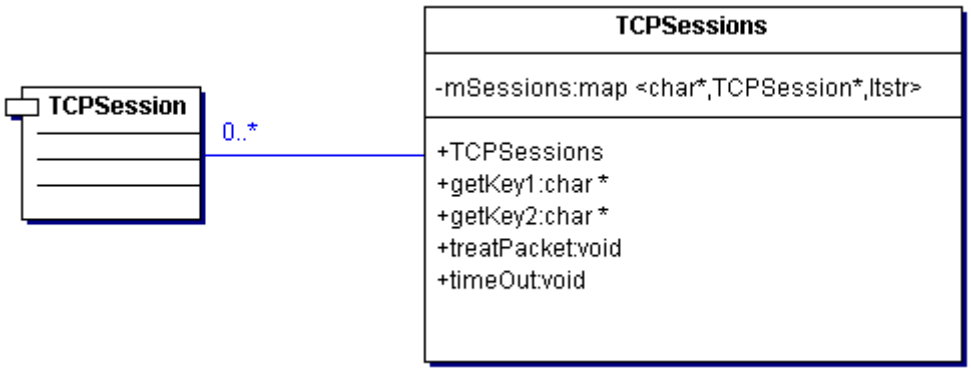
```
map <u_short, IPFragments*>::iterator iter;
bool fragment = ipPacket->getIsFragment();
IPFragments* frag;
typedef pair <u_short,IPFragments*> pairMap;
TCPPacket* tcpPacket;

if(!fragment){
//si no esta fragmentado se hace el paquete TCP correspondiente
  al paquete IP
  tcpPacket = new TCPPacket(ipPacket);
}else{  //si esta fragmentado
  u_short key = ipPacket->getIdFragment();
  iter = stack.find(key);
  if (iter == stack.end()){
       frag = new IPFragments(ipPacket);
       stack.insert(pairMap(key,frag));
        tcpPacket = NULL;
  }else{
       frag = (*iter).second;
       tcpPacket = frag->insert_fragment(ipPacket);
        if (tcpPacket!=NULL) stack.erase(iter);
        else (*iter).second = frag;
  }
}
return tcpPacket;
```

6.6 Clase TCPSessions

- **Funcionamiento general y objetivos de la clase.**

El objetivo fundamental de esta clase es el de almacenar todas las sesiones activas entre el cliente y el servidor. Esta clase posee una *tabla hash* en la cual cada entrada es un objeto TCPSession que gestiona una sesión. Todos los paquetes capturados pertenecen a alguna sesión por tanto todos tienen que pasar por el tratamiento de esta clase. Cuando un paquete TCP es tratado por esta clase se busca si pertenece a una sesión ya creada, de ser así se inserta en esa sesión y se trata. Si no pertenece a una sesión ya iniciada se crea una nueva entrada en la *tabla hash* para una nueva sesión.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **mSessions:** este atributo es una *tabla hash* que almacena todas las sesiones activas. Para hacer una clave que identifique a las sesiones unívocamente hay que tener en cuenta el sentido de la conexión. Para cada sentido hay una clave y estas se tienen que buscar para ver si alguna de las dos está en la tabla. Si no se haya ninguna de las dos claves se crea una sesión y una nueva entrada en la tabla donde insertar la sesión. Si existe alguna entrada para alguna de las dos claves se inserta el paquete en la sesión de la entrada encontrada.

  - **TCPSessions():** este método es el constructor de la clase que inicializa *tabla hash* a vacía.

- **char* getKey1(TCPPacket* paq):** este método genera una forma posible de clave para una sesión, es decir, la genera en el sentido fuente-destino (ip fuente+ip destino+puerto fuente+puerto destino).
- **char* getKey2(TCPPacket* paq):** este método genera la otra forma posible de clave para una sesión, es decir, la genera en el sentido destino-fuente (ip destino+ip fuente+puerto destino+puerto fuente).

  Tanto para key1 y key2, la clave se calcula de la misma forma. Para key1:

```
char* key;
char* sIP = paq->getSourceIP();
char* tIP = paq->getTargetIP();
char* sPort = paq->getSourcePort();
char* tPort = paq->getTargetPort();

key=(char*)malloc(strlen(sIP)+strlen(tIP)+strlen(sPort)+strlen(tPort)+1);
if (key == NULL){
    printf("error TCPSessions/getKey1:: memory failed in key\n");
    fflush(stdout);
    exit(-1);
}
key = strcpy(key,"");
key = strncat(key,sIP,strlen(sIP));
key = strncat(key,tIP,strlen(tIP));
key = strncat(key,sPort,strlen(sPort));
key = strncat(key,tPort,strlen(tPort));
key = strncat(key,"\0",1);

return key;
```

- **void treatPacket(TCPPacket* paqTCP):** este método trata los paquetes que le llegan insertándolos en la sesión que le pertenece si ya existe, o creando una nueva si no pertenecen a ninguna de las ya creadas.

```
char* key1 = getKey1(paq);
char* key2 = getKey2(paq);
map <char*,TCPSession*,ltstr>::iterator iter;
typedef pair <char*,TCPSession*> pairS;
TCPSession* ses;

iter = mSessions.find(key1);
if (iter == mSessions.end()){//si no esta con la key1 se prueba
                              con la key2
    iter = mSessions.find(key2);
    if (iter == mSessions.end()){//si no esta con la key1 y key2
                                  se crea una nueva sesion
        TCPSession* ses = new TCPSession(paq);
```

```
        mSessions.insert(pairS(key2,ses));
    }else{//si esta con la key2
          ses = (*iter).second;
          ses->treatPacket(paq);
          (*iter).second = ses;
     }
}else{//si esta con la key1
      ses = (*iter).second;
      ses->treatPacket(paq);
      (*iter).second = ses;
}
```

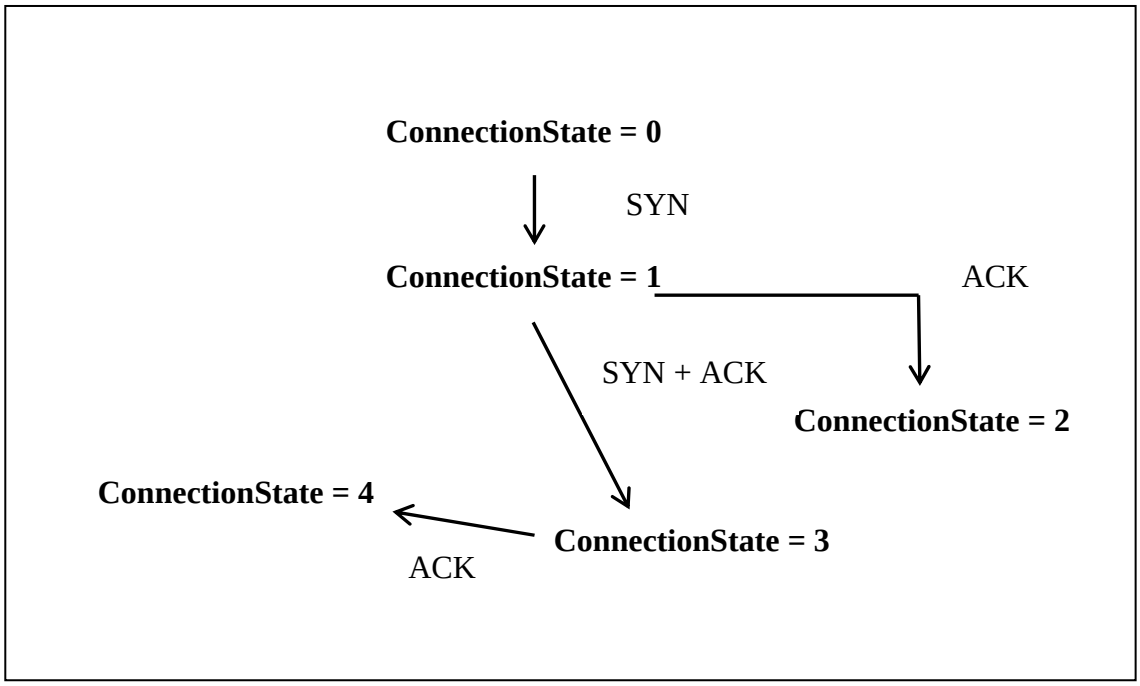
6.7 Clase TCPSession

- **Funcionamiento general y objetivos de la clase.**

  El objetivo principal de esta clase es simular una conversación a nivel TCP entre dos máquinas. Contiene 2 atributos del tipo Window que simulan las ventanas de conversación TCP en cada sentido, así como un atributo del tipo HTTPSession que será el encargado de ir almacenando la conversación a nivel http.

  Establecimiento de la conexión: El procedimiento de tres pasos se utiliza para establecer una conexión. Normalmente, este procedimiento se inicia por un TCP y responde otro TCP distinto. El procedimiento también funciona si dos TCP simultáneamente inician el procedimiento. En el caso de intentos simultáneos, cada TCP recibe un segmento 'SYN', que no lleva acuse de recibo, tras haber envíado su 'SYN'. Por supuesto, la llegada de un segmento 'SYN' anterior duplicado puede, potencialmente, hacer creer al receptor que está en progreso una iniciación simultánea de conexión. Un uso adecuado de los segmentos de tipo 'reset' puede eliminar la ambigüedad de estos casos.

  Antes de que dos máquinas empiecen a comunicarse necesitan establecer una conexión entre ellas mediante el procedimiento de acuerdo en tres pasos o hand-shake. Normalmente, este procedimiento se inicia por un TCP y responde otro TCP distinto, aunque también funciona si dos TCP simultáneamente inician el procedimiento. Este mecanismo se ha simulado con un sistema de estados de la variable *connectionState*.



Cierre de la conexión: Existen fundamentalmente tres casos:

1) El usuario inicia el cierre de la conexión enviando la llamada CLOSE a TCP.
2) El TCP remoto inicia el cierre enviando una señal de control FIN.
3) Ambos usuarios realizan la llamada CLOSE simultáneamente.

Caso 1: el usuario local inicia el cierre

En este caso se puede construir un segmento FIN y ponerlo en la cola de salida de segmentos. TCP no aceptará más llamadas SEND y entrará en el estado FIN-WAIT-1. En este estado se permiten las llamadas RECEIVE. Todos los segmentos que preceden al FIN, incluído éste, serán retransmitidos hasta que se reciban los correspondientes acuses de recibo. Cuando el TCP remoto haya realizado el acuse de recibo del FIN y enviado su propio FIN, el TCP local puede realizar el acuse de recibo de este FIN. Nótese que un TCP que recibe un FIN enviará su ACK pero no enviará su propio FIN hasta que su usuariohaya cerrado también la conexión con un CLOSE.

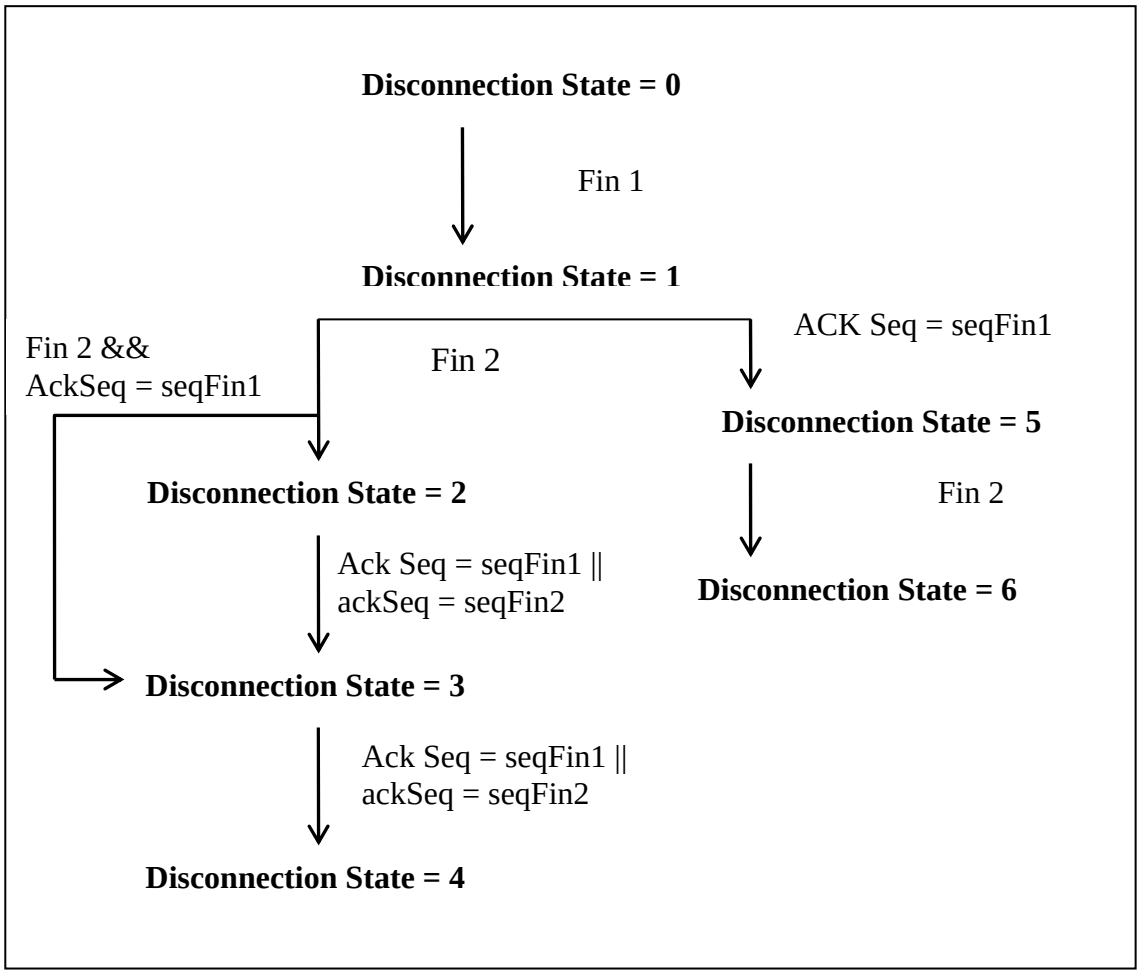
Caso 2: TCP recibe un FIN desde la red

Si llega un FIN no solicitado desde la red, el TCP receptor puede responder con un ACK y advertir al usuario de que la conexión se está cerrando. El usuario responderá con una llamada CLOSE, ante la cual TCP puede enviar un FIN al otro TCP tras enviar los datos restantes. Entonces TCP espera hasta el acuse de recibo de su propio FIN y elimina la conexión. Si el ACK no llega en el tiempo de espera del usuario, la conexión se aborta y así se notifica al usuario.

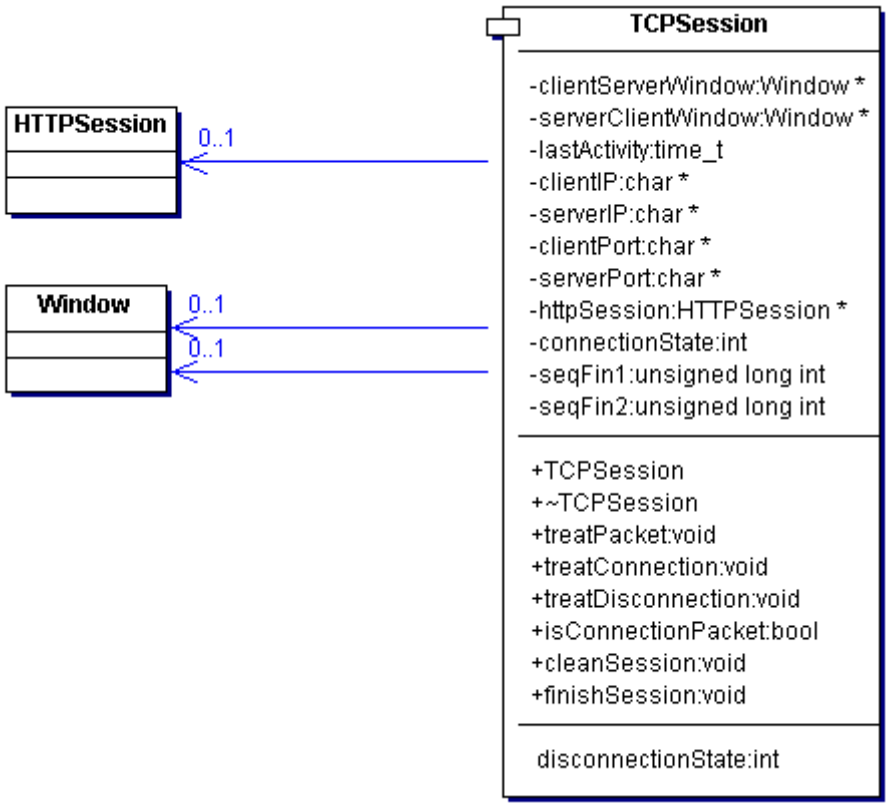
Caso 3: ambos usuarios cierran simultáneamente

El cierre simultáneo a ambos lados de la conexión causa el intercambio de segmentos FIN. Cuando todos los segmentos que preceden a los FIN se han procesado y confirmado con acuses de recibo, cada TCP puede responder con un ACK el FIN que ha recibido. Ambos, ante la recepción de los ACK, eliminarán la conexión.

Para la desconexión se sigue también un diagrama de estados. El usuario que hace la llamada a close puede continuar recibiendo datos mediante llamadas receive hasta que se le indique que el otro lado ha cerrado también la conexión.
Este mecanismo se ha simulado con un sistema de estados de la variable *connectionState*.



- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **ClientServerWindow**: Ventana que simula la conversación TCP en el sentido servidor - cliente. Almacena los paquetes TCP llegados del servidor que contendrán las respuestas a peticiones del cliente.
  - **ServerClientWindow**: Ventana que simula la conversación TCP en el sentido cliente - servidor. Almacena los paquetes TCP enviados por el cliente que contendrán peticiones del mismo a un determinado servidor.
  - **ClientIP, clientPort,serverIP, serverPort:** Almacenan la IP del cliente y servidor, así como sus respectivos puertos. La sesión se identifica unívocamente por el valor de estos cuatro campos, ya que entre 2 mismas IP's puede establecerse más de una conexión TCP, pero es improbable que éstas reutilicen un puerto en la máquina del cliente usado con anterioridad en un breve espacio de tiempo.
  - **ConnectionState:** variable entera que representa el estado en el que se encuentra el establecimiento de la conexión TCP.
  - **DisconnectionState:** variable entera que representa el estado en el que se encuentra la desconexión TCP.
  - **SeqFin1, seqFin2:** variables que almacenan los números de secuencia de los paquetes TCP de FIN tanto del cliente como del servidor, para

detectar confirmaciones futuras a esos FIN mediante paquetes ACK de confirmación.

- **LastActivity:** indica el tiempo que lleva esta sesión TCP sin actividad. Útil para cerrar sesiones cuando el protocolo de fin de conexión no se cumple o hay pérdidas de paquetes. Cuando este tiempo sea mayor al valor máximo de inactividad fijado, la sesión se dará por terminada.
- **TCPSession(TCPPacket* paq):** Una sesión TCP siempre se crea con un paquete TCP de SYN proveniente del cliente, creando las 2 ventanas de simulación de conversación TCP entre el cliente y servidor, una en cada sentido, la sesión HTTP e inicializando las restantes variables con sus valores adecuados.

```
connectionState = 0;
disconnectionState = 0;
seqFin1 = 0;
seqFin2 = 0;

if ((paq->getSyn()) && (!paq->getAck())){
   if (time(&lastActivity) == ((time_t)-1) ){
       printf("error TCPSession/TCPSession:: error at getting time
              in TCPSession\n");
       exit(-1);
   }

   connectionState = 1;
   httpSession = new HTTPSession();
   clientServerWindow = new Window();
   serverClientWindow = new Window();

    char* aux = paq->getSourceIP();
    int length = strlen(aux);
    clientIP = (char*)malloc(length+1);
    if (clientIP == NULL){
       printf("error TCPSession/TCPSession:: memory failed in
              clientIP\n");
       fflush(stdout);
       exit(-1);
   }
   bcopy(aux,clientIP,length);
   clientIP[length] = '\0';

   aux = paq->getTargetIP();
   length = strlen(aux);
   serverIP = (char*)malloc(length+1);
   if (serverIP == NULL){
     printf("error TCPSession/TCPSession:: memory failed in
            serverIP\n");
     fflush(stdout);
     exit(-1);
```

```
   }
   bcopy(aux,serverIP,length);
   serverIP[length] = '\0';

   aux = paq->getSourcePort();
   length = strlen(aux);
   clientPort= (char*)malloc(length+1);
   if (clientPort == NULL){
     printf("error TCPSession/TCPSession:: memory failed in
             clientPort\n");
     fflush(stdout);
     exit(-1);
   }
   bcopy(aux,clientPort,length);
   clientPort[length] = '\0';

   aux = paq->getTargetPort();
   length = strlen(aux);
   serverPort= (char*)malloc(length+1);
   if (serverPort == NULL){
     printf("error TCPSession/TCPSession:: memory failed in
     serverPort\n");
     fflush(stdout);
     exit(-1);
   }
   bcopy(aux,serverPort,length);
   serverPort[length] = '\0';

   numSessionOpen ++;
}
else{
  printf("error TCPSession/TCPSession:: a TCPSession must be
         created by a synchronized TCPPacket\n");
}
```

- **void treatConnection(TCPPacket* paq):** Método que implementa el diagrama de estados de conexión especificado previamente.

```
if ((connectionState == 1) && paq->getSyn() && paq->getAck())
     connectionState = 3;
else if ((connectionState == 1) && paq->getAck())
     connectionState = 2;
else if ((connectionState == 2) && (paq->getSyn()))
   connectionState =3;
else if ((connectionState == 3) && (paq->getAck()))
     connectionState =4;
```

- **void treatDisconnection(TCPPacket* paq):** Método que implementa el diagrama de estados de conexión especificado previamente.

```
if ((disconnectionState == 0) && paq->getFin()){
   disconnectionState = 1;
   seqFin1=paq->getSeq()+(unsigned long int)paq->getDataLength();
}
else if ((disconnectionState == 1) && paq->getFin() &&
          paq->getAck() && (paq->getAckSeq() == (seqFin1 + 1))){
   disconnectionState = 3;
   seqFin2=paq->getSeq()+(unsigned long int)paq->getDataLength();
}
else if ((disconnectionState == 1) && paq->getFin()){
   disconnectionState = 2;
   seqFin2=paq->getSeq()+(unsigned long int)paq->getDataLength();
}
else if ((disconnectionState == 2)&& paq->getAck() &&
         ((paq->getAckSeq() == seqFin1+1) ||
         (paq->getAckSeq() == seqFin2+1)))
   disconnectionState = 3;

else if ((disconnectionState == 3) && paq->getAck() &&
         ((paq->getAckSeq() == seqFin1+1) ||
          (paq->getAckSeq() == seqFin2+1)))
   disconnectionState = 4;

else if ((disconnectionState == 1) && paq->getAck() &&
         (paq->getAckSeq() == seqFin1+1))
   disconnectionState = 5;

else if ((disconnectionState == 5) && paq->getFin()){
   disconnectionState = 6;
   seqFin2 paq->getSeq()+(unsigned long int)paq->getDataLength();
}
else if ((disconnectionState == 6) && (paq->getAck()) &&
         (paq->getAckSeq() == seqFin2+1)){
   disconnectionState = 4;
}
makeDisconnection();
```

- **void treatPacket(TCPPacket* paq):** Cuando llega un nuevo paquete TCP a esta conexión, lo primero que se debe hacer es comprobar que el handshake se ha realizado correctamente (connectionState == 3). Si aún la conexión no está establecida, se invocará al método treatConnection que gestiona este establecimiento.

  Si existe conexión, se analiza el paquete entrante comprobando su sentido de envío, insertándolo en la ventana que le corresponda, y obteniendo el posible fragmento http que confirme su ACK en la ventana opuesta.

  El caso especial es que sea un paquete con el flag RESET activado, por lo que la conexión se termina en ese mismo momento.

```
if (httpSession == NULL){
 /* The session is finish. */
 return;
}

if (paq->getRst() == true){
  cleanSession();
}
else{
 if (isConnectionPacket(paq)){
   treatConnection(paq);
 }
 else{
   if(disconnectionState == 4){
   }else{
     if (connectionState == 4){
        char* sourceIP, *targetIP;
        sourceIP = paq->getSourceIP();
        targetIP = paq->getTargetIP();
        if (strcmp(sourceIP,clientIP) == 0){
           if (paq->getAck() == true){
             unsigned long int ack = paq->getAckSeq();
             clientServerWindow->insert(paq);
             u_short len=0;
             u_char* http = serverClientWindow->getHttp(ack,&len);
             if (len != 0){
               httpSession->insertResponse(http,len);
             }
           }
         }else{
           if (strcmp(targetIP,clientIP) == 0){
             if (paq->getAck() == true){
               unsigned long int ack = paq->getAckSeq();
               serverClientWindow->insert(paq);
               u_short len=0;
              u_char*http=clientServerWindow->getHttp(ack,&len);
               if (len != 0){
                    httpSession->insertRequest(http,len);
               }
             }
           }else{
              printf("error TCPSession/treatPacket:: trying to
                  treat a packet that isn't of this session.\n");
           }
         }
       if (isDisconnectionPacket(paq))
          treatDisconnection(paq);
       } else{


          printf("error TCPSession/treatPacket:: trying to treat
                 a data packet before making a connection.\n");
       }
```

```
    } //else disconnectionState
  }//else isConnectionPacket
}//else getRst
```

- **void cleanSession():** método invocado cuando una sesión TCP termina y cuya finalidad es liberar todo la memoria reservada para tratar la sesión.

```
if (clientServerWindow != NULL)
   delete clientServerWindow; clientServerWindow = NULL;
if (serverClientWindow != NULL)
   delete serverClientWindow; serverClientWindow = NULL;
if (clientIP != NULL)
   free(clientIP); clientIP = NULL;
if (serverIP != NULL)
   free(serverIP); serverIP = NULL;
if (clientPort != NULL)
  free(clientPort); clientPort = NULL;
if (serverPort != NULL)
   free(serverPort); serverPort = NULL;
if (httpSession != NULL)
   delete httpSession; httpSession = NULL;
```

- **bool isConnectionPacket(TCPPacket* paq):** método que devuelve un booleano indicando si el paquete TCP tiene activado algún flag de conexión o no.

```
if (paq->getSyn()) return true;
else if (paq->getAck() && (connectionState == 1)) return true;
else if (paq->getAck() && (connectionState == 3)) return true;
return false;
```

- **bool isDisconnectionPacket(TCPPacket* paq):** devuelve un booleano si se trata de un paquete de FIN.

```
if (paq->getFin()) return true;
else if (paq->getAck() && (disconnectionState != 0))return true;
return false;
```

- **int getDisconnectionState():** método que devuelve el estado de la desconexión.
- **void makeDisconnection():** Si la sesión se ha terminado correctamente mediante el protocolo de fin de conexión, se debe eliminar de la lista de

sesiones en curso y comprobar si existe un único paquete http en su sesión http, tratándolo en caso positivo.

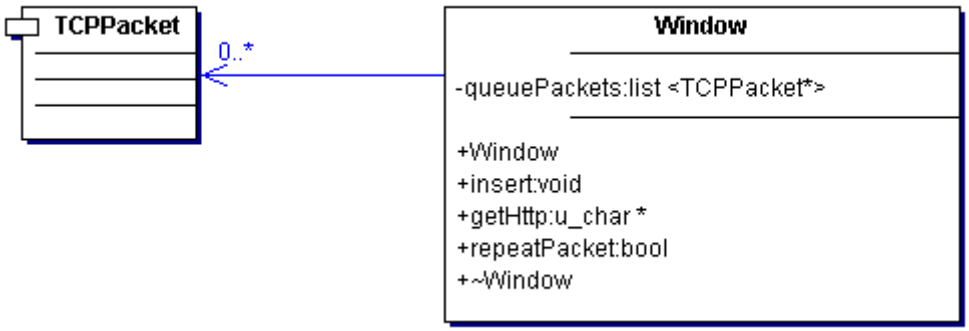
```
if (disconnectionState == 4){
  if (httpSession!= NULL){
       numSessionClose ++;
       httpSession->obtainPacket();
       cleanSession();
  }
}
```

6.8 Clase Window

- **Funcionamiento general y objetivos de la clase.**

El objetivo principal de esta clase es el de simular el mecanismo de ventana deslizante TCP. Realiza el tratamiento en sólo un sentido. Inserta los paquetes y los ordena por el número de secuencia. Cuando se produce una confirmación de por parte del otro extremo de la conexión esta clase devuelve el contenido HTTP confirmado y lo elimina de la ventana.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **queuePackets:** lista de paquetes TCP ordenados por su número de secuencia. Estos paquetes no han sido todavía confirmados por el otro extremo de la conexión. En esta lista no existen paquetes repetidos.

  - **Window():** constructor de la clase que inicializa la lista a vacía.
  - **~Window():** destructor de la clase que libera toda la memoria utilizada.
  - **bool repeatPacket(TCPPacket* paq):** este método devuelve true si el paquete que se pasa por parámetro es un paquete que ya está en la lista de paquetes, es decir, es un paquete repetido. Paquetes repetidos se consideran a los que tienen igual número de secuencia, número de confirmación (*ack*) y longitud de los datos que contiene. Devuelve false en caso contrario.

```
list <TCPPacket*>::iterator l1;
TCPPacket* paqAux;
l1 = queuePackets.begin();
int i=0;
```

```
bool end = false;
bool equals = false;
unsigned long int ack,seq;

while(l1 != queuePackets.end() && !end && !equals){
      paqAux = (*l1);
      ack = paqAux->getAckSeq();
      seq = paqAux->getSeq();
       if (ack == paq->getAckSeq() && seq == paq->getSeq() &&
           (*l1)->getDataLength()==paq->getDataLength())
                    equals = true;
       else if (seq > paq->getSeq())
                end = true;
            else l1++;
}
return equals;
```

- **void insert(TCPPacket* tcpPaq):** este método inserta de forma ordenada un paquete en la lista si no está repetido. Si está repetido este método no hace nada.
- **u_char* getHttp(unsigned long int ack, u_short* lenDataHTTP):** este método toma como parámetro un número de confirmación (ack) y devuelve, si existen, los datos HTTP y la longitud de los datos de los paquetes confirmados. Es decir, si un *ack* confirma varios paquetes de la lista *queuePackets* el método devuelve la concatenación ordenada de los datos HTTP y la longitud de estos en *lenDataHTTP*. Si el *ack* no confirma a ninguno de los paquetes de la lista o ninguno de los paquetes confirmados tiene datos HTTP se devuelve NULL y *lenDataHTTP* cero. Obviamente los paquetes que no contienen datos HTTP no influyen en lo que devuelve la función. Cuando un paquete es confirmado (ya tenga o no datos HTTP) todos los paquetes anteriores a él y él son eliminados para después formar las respuestas del método. Si en la lista hay varios paquetes consecutivos con el mismo número de secuencia se confirma el paquete de la posición más alta de la lista.

```
while(l1!=queuePackets.end()){
    if ((*l1)->getFin() || (*l1)->getSyn()){
       ackAux = ack-1;
    }else ackAux = ack;
    if ((*l1)->getDataLength()==0) {
        if ((*l1)->getSeq()==ackAux){
              stop=true;
        }
        else{
          if(stop){
              break;
         }
```

```
      }
    }else{
       unsigned long int aux=(*l1)->getSeq();
       unsigned long int aux2= (*l1)->getDataLength();
      if((*l1)->getSeq()+(unsigned long int)(*l1)->
          getDataLength() ==ackAux){
        existData=true;
        stop=true;
      }else{
        if (stop){
             break;
        }
      }
    }
    l1++;
}
if (existData==true){
  for (l2=queuePackets.begin();l2!=l1;l2++){
       httpLength=httpLength+(*l2)->getDataLength();
  }
  data=(u_char*)malloc(httpLength);



  if (data == NULL){
       printf("Error Window/getHttp:: memory failed in data\n");
       exit(-1);
  }
  for (l2=queuePackets.begin();l2!=l1;l2++){
      //confirmo hasta l1 este no incluido
        if ((*l2)->getDataLength()!=0){
            dataAux = (u_char *)((*l2)->getData());
            memcpy((u_char*)data,dataAux,(*l2)->getDataLength());
            data=data+(*l2)->getDataLength();
         }
  }
  l2=queuePackets.begin();
  while(l2 != l1){
        delete (*l2);
        *l2 = NULL;
        l2 ++;
  }
  l2=queuePackets.begin();
  queuePackets.erase(l2,l1);
  data=data-httpLength;
  *lenDataHTTP = httpLength;

}else{//si no hay datos
   if (stop){ //si no se ha llegado al final es decir existe
                algun ack confirmado
       l2=queuePackets.begin();
       while(l2 != l1){
            delete (*l2);
            *l2 = NULL;
```

```
            l2 ++;
        }
        l2=queuePackets.begin();
        queuePackets.erase(l2,l1);
    }
return data;
```

6.9 Clase HTTPSession

- **Funcionamiento general y objetivos de la clase.**

El objetivo de esa clase es reconstruir la conversación HTTP a medida que se van procesando paquetes TCP por la ventana TCP. Esta conversación siempre se realiza por sucesivas peticiones HTTP por parte del cliente a uno o varios servidores. El protocolo http permite varios tipos de peticiones, siendo los más frecuentes GET, POST y HEAD.

El detalle principal a tener en cuenta es que un cliente no realizará una segunda petición, ya sea al mismo servidor o a otro, hasta que no haya obtenido la respuesta completa a su petición anterior. Por tanto, se debe controlar el fin de la petición, y sobre todo el fin de la respuesta a esta petición.

Se tienen cuatro opciones para detectar el final de una respuesta a una petición http:

- **Transfer Encoding:** Si un servidor quiere empezar a mandar la respuesta antes de conocer su longitud total puede usar el tipo de codificación de transferencia *chunked* fijando la cabecera Transfer-Encoding a chunked. Asi, fraccionará la respuesta completa en pequeños trozos y los mandará en serie.
  El cuerpo de un mensaje chunked se compone de una serie de fragmentos seguidos por un cero, mas unas cabeceras opcionales y una línea en blanco. Cada chunk consiste en dos partes :
  - Una línea con la longitud de datos del fragmento, posiblemente seguido por un punto y coma y unos parámetros opcionales que se pueden ignorar, terminando en un CRLF.
  - Los propios datos, seguidos por un CRLF.

  Así, una respuesta chunked tiene un aspecto similar al siguiente:

```
http/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

1a; todo lo que va aqui se debe ignorer
abcdefghijkmnopqrstuvwz
10
123456789abcdef
```

0
cabeceraOpcional1: valor1
cabeceraOpcional2: valor2
[línea en blanco]

Como comparación, la respuesta anterior es equivalente a esta:

http/1.1 200 OK
Content-Type: text/plain
Content-Length: 42
cabeceraOpcional1: valor1
cabeceraOpcional2: valor2

abcdefghijkmnopqrstuvwz123456789abcdef

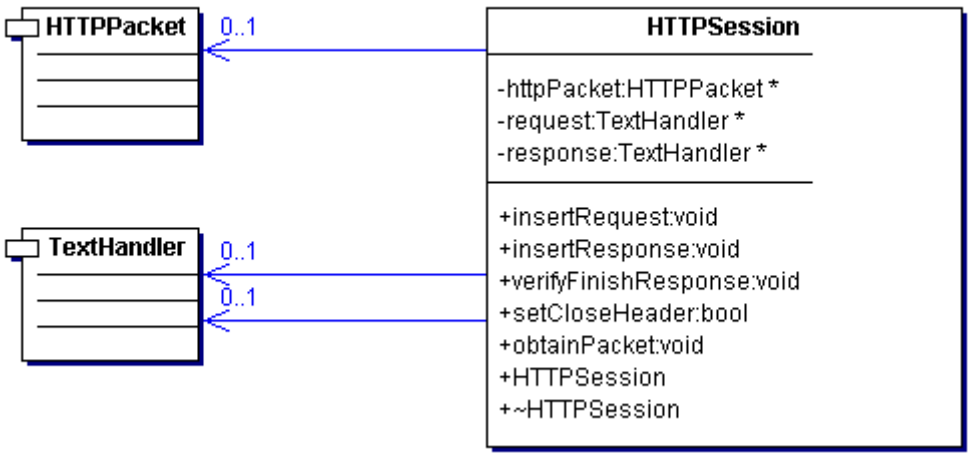
- **Content Length:** Algunos servidores facilitan la cabecera Content-Length en su respuesta que indica, en bytes, la dimensión del cuerpo de la misma. Sabiendo el tamaño de la respuesta, se puede dar por terminada una petición si ese valor coincide con el tamaño del atributo response, que lleva acumulada los bytes de respuesta recibidos.
- **Content Type:** Si no estamos en ninguno de los dos caso anteriores, pero el cliente ha pedido una página html, se puede localizar el fin de la respuesta buscando el tag de cierre de la página pedida </html>
- **Siguiente petición:** Siempre, y como último caso, se sabe que la respuesta ha terminado cuando el cliente realice una siguiente petición.
- **Cierre conexión:** Si la conexión se cierra por ella ya no se podrán realizar más peticiones, por lo que se debe liberar la memoria reservada y recuperar la posible petición y respuesta pendientes.

Ya que tanto la petición como la respuesta http puede y suelen venir en varios paquetes TCP, esta clase tiene 2 objetos de tipo TextHandler para ir almacenando toda la conversación http. Cuando se detecte el fin de una petición (respuesta) estas variables se inicializarán de nuevo y se actualizará el objeto de la clase HTTPPacket con la petición (respuesta) completa.

Además, posee un objeto de tipo EventHandler que gestiona los diferentes eventos que se dan en la sesión TCP. Un evento se ocasiona tanto directa como indirectamente por el usuario. La navegación Web genera un evento cada vez que el

cliente pincha un nuevo enlace, participando directamente en la creación de eventos, así como cada vez que se abre un pop up de publicidad.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **httpPacket:** atributo de tipo HTTPPacket que almacena **una** petición y su correspondiente respuesta de forma estructurada y fácil de acceder.
    Este objeto almacena únicamente una petición y su respuesta, y una vez completas ambas, se invocará al método insert de event handler, que enlazará el paquete http completado con la sesión de navegación a la que pertenece.
  - **request:** atributo de tipo TextHandler que gestiona todo lo referente al parseo del texto HTTP de la petición, acumulando los bytes de petición recibidos hasta que llegue la petición completa.
  - **response:** atributo de tipo TextHandler que gestiona todo lo referente al parseo del texto HTTP de la respuesta, acumulando los bytes de respuesta recibidos hasta que llegue la respuesta completa, momento en el que se enlazará el paquete http completo recibido con la sesión de navegación a la que pertenece y se inicializarán los atributos request, response, y httpPacket preparándolos para la siguiente petición.

  - **HTTPSession():** constructor de la clase el cual inicializa los atributos.
  - **void insertRequest(u_char* data, u_short lenDataHTTP):** Método al cual se invoca cuando se reconstruye conversación http. Insertará los bytes correspondientes como parte de la petición.

```
if (httpPacket->getHeaderRequest() != NULL){
    httpPacket->putBodyResponse(response->getData(-1),
                                response->getDataLength());
    httpPacket->reset();
    request->cleanData();
    response->cleanData();
}
request->insertData(data,lenData);
if(httpPacket->getHeaderRequest() == NULL){
   char* cabecera = request->getHeader();

   if (cabecera != NULL){
       httpPacket->putHeaderRequest(cabecera);
       bool chunked = httpPacket->requestIsChunked();
       int requestDataLength = httpPacket->getRequestDataLength();
       if (chunked) {
         char* blankLine = "\r\n\r\n";
         if (request->finishIn(blankLine))
            httpPacket->putBodyRequest(request->getData(-1),
                                       request->getDataLength());
         }else {
           if (requestDataLength != -1){
            u_char* body = request->getData(requestDataLength);
             if (body != NULL){
               httpPacket->putBodyRequest(body,requestDataLength);
             }
           }
         }
      }//chuncked
   }else{
       bool chunked = httpPacket->requestIsChunked();
       int requestDataLength = httpPacket->getRequestDataLength();
       if (chunked) {
         char* blankLine = "\r\n\r\n";
         if (request->finishIn(blankLine))
            httpPacket->putBodyRequest(request->getData(-1),
                                       request->getDataLength());
         }else {
           if (requestDataLength != -1){
            u_char* body = request->getData(requestDataLength);
             if (body != NULL){
               httpPacket->putBodyRequest(body,requestDataLength);
             }
           }
         }
       }
    }
}
```

- **void insertResponse(u_char* data, u_short lenDataHTTP):** Método al que se invoca cuando se reconstruye conversación http. Insertará los bytes correspondientes como parte de la respuesta http e invocará al

método verifyFinishResponse para comprobar si la respuesta ha finalizado o no.

```
response->insertData(data,lenData);

if(httpPacket->getHeaderResponse() == NULL){
  char* cabecera = response->getHeader();
  if (cabecera != NULL){
    char* token = "HTTP/";
    char* pos = strstr(cabecera,token);
    char* type;
    if (pos != NULL){
       pos = pos + 9;
      int value = strncmp(pos,"1",1);
      if (value == 0){
         response->cleanData();
      }else{
         httpPacket->putHeaderResponse(cabecera);
         verifyFinishResponse();
      }
     }// if pos != NULL
  }// if cabecera != NULL
} //if httpPacket->getHeaderResponse() == NULL

else{
  verifyFinishResponse();
}
```

- **void verifyFinishResponse():** Método que comprueba si la respuesta http ha llegado completa o no. De estarlo, se enlaza el paquete http completo recibido con la sesión de navegación a la que pertenece y se inicializarán los atributos request, response, y httpPacket preparándolos para la siguiente petición..

```
bool chunked = httpPacket->responseIsChunked();
int responseDataLength = httpPacket->getResponseDataLength();
if (chunked) {
   char* blankLine = "\r\n\r\n";
   if (response->finishIn(blankLine)){
    httpPacket->putBodyResponse(response->getData(-1),
                    response->getDataLength());
      httpPacket->reset();
      request->cleanData();
      response->cleanData();
    } // if FinishIn
} // if chuncked
else {
  if (responseDataLength != -1){
```

```
     u_char* body = response->getData(responseDataLength);
     if (body != NULL){
         httpPacket->putBodyResponse(body,responseDataLength);
         httpPacket->reset();
         request->cleanData();
         response->cleanData();
     }
   } //if responseDataLength
   else{
       if (httpPacket->responseIsHtml()){
          char* tag = "</html>";
          if (response->containTag(tag)){
            httpPacket->putBodyResponse(response->getData(-1),
                                        response->getDataLength());
            httpPacket->reset();
            request->cleanData();
            response->cleanData();
          }
        } // if response is html
   } // else responseDataLength != -1
}// else not chuncked
```

- **bool setCloseHeader():** Método que devuelve true si, o bien el cliente, o bien el servidor, han incluido la cabecera Connnection: Close. Devuelve false en otro caso.
- **void obtainPacket():** Este método sirve para capturar aquella conversación http (par petición, respuesta) que son las últimas en una sesión TCP y que no se han podido detectar su final mediante ninguno de los métodos descritos previamente.

```
if (httpPacket == NULL){
    printf("error HTTPPacket/obtainPacket::
            httpPacket is null\n");
    exit(-1);
}


if (httpPacket->getBodyResponse() == NULL &&
         httpPacket->getHeaderRequest() != NULL){
  httpPacket->putBodyResponse(response->getData(-1),
                              response->getDataLength());
  httpPacket->reset();
}
```

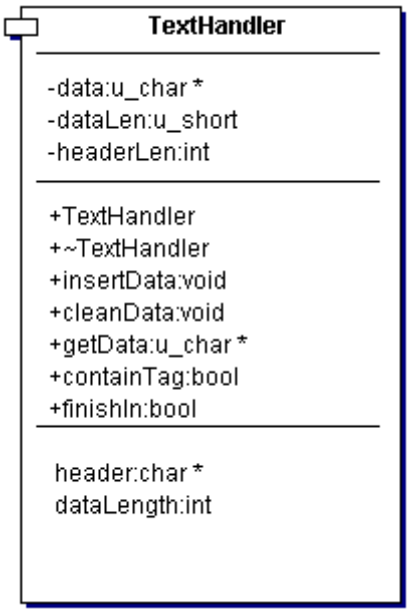
- **HTTPPacket* getHttpPacket():** devuelve el valor el atributo httpPacket.

6.10 Clase TextHandler

- **Funcionamiento general y objetivos de la clase.**

  El objetivo de esta clase es el almacenamiento efectivo de caracteres que representarán parte de una respuesta o petición http, formada por cabecera y cuerpo, y sus correspondientes funciones de acceso.
  Tanto la cabecera como la respuesta pueden ser de diferentes tamaños y contener información con distintos formatos. Las cabeceras de las peticiones como de las respuestas http siempre son String y terminan en una línea en blanco, sin embargo, el cuerpo de la respuesta puede venir en formatos muy variados.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **data:** array dinámico que almacena todos los caracteres recibidos.
  - **dataLen:** entero que indica la longitud del array anterior.
  - **headerLen:** entero que indica el número de bytes dentro del array data que constituyen la cabecera.

  - **TextHandler():** constructor que inicializa las variables de la clase.
  - **bool containTag(char* token):** método que devuelve un bool indicando si el token que se le pasa como parámetros está presente en data.

```
if (dataLen!=0){
   int tokenLen = strlen(token);
   u_char* pos = data + headerLen;
   char* find = strstr((char*)pos,token);
   if ((find == NULL) || (find - (char*)data > dataLen))
       return false;
   else return true;
}else{
   printf("error TextHandler/containTag:: dataLen is 0\n");
   fflush(stdout);
   exit(-1);
}
```

- **char* getHeader():** método que devuelve la cabecera almacenada o NULL en el caso de que esta no haya llegado completa. El fin de una cabecera se detecta porque siempre termina en una línea en blanco (/r/n/r/n).

```
if (dataLen!=0){
  char* blankLine = "\r\n\r\n";
  char* pos = strstr((char*)data,blankLine);
  if (pos == NULL || (pos - (char*)data > dataLen)){
     return NULL;
  }else{
    if (pos - (char*)data > dataLen) {
      printf("error TextHandler/getHeader::
              getHeader out of memory\n");
      fflush(stdout);
      exit(-1);
    }
    pos = pos+4;
    headerLen = pos - (char*)data;
    if (headerLen > dataLen){
      printf("error TextHandler/getHeader::
                    headerLen > dataLen\n");
      exit(-1);
    }
    char *header = (char*)malloc(headerLen+1);
    if (header == NULL){
      printf("error TextHandler/getHeader:: memory failed
                    in header\n");
      fflush(stdout);
      exit(-1);
    }
    memcpy(header,data,headerLen);
    header[headerLen] = '\0';
    return header;
  }
}else{//dataLen ==0
   printf("error TextHandler/getHeader:: dataLen is 0\n");
```

```
   fflush(stdout);exit(-1);
}
```

- **u_char* getData(int numBytes):** método que devuelve bytes de data o NULL dependiendo del valor del parámetro numBytes.

```
if (dataLen!=0){
  if (numBytes == -1){
     return (data + headerLen);
  }else{
    if (numBytes == (dataLen-headerLen)){
        return (data+headerLen);
     } else return NULL;
  }
}else{
  return NULL;
}
```

- **bool finishIn(char* token):** método que comprueba que los últimos caracteres almacenados en data coinciden con el token que se le pasa como parámetro.

```
if (dataLen!=0){
  int tokenLen = strlen(token);
  u_char* pos = data + dataLen - tokenLen;
  char* find = strstr((char*)pos,token);
  if ((find == NULL) || (find - (char*)data > dataLen))
     return false;
  if (find - (char*)data > dataLen) {
      printf("error TextHandler/finishIn:: out of memory\n");
      fflush(stdout);
      exit(-1);
  }else return true;
}else{
   printf("error TextHandler/finishIn:: dataLen is 0\n");
   fflush(stdout);
   exit(-1);
}
```

- **void cleanData():** libera la memoria reservada en data y prepara los atributos para recibir más datos de siguientes peticiones.
- **void insertData(u_char* d,u_short lenDataHTTP):** Los paquetes HTTP tienen tamaños diversos por lo que para una asignación óptima de la memoria hay que asignarla dinámicamente a la variable data según vayan llegando fragmentos http.

```
int dataLenAux;
u_char* dataAux;
if (lenDataHTTP >0 && d != NULL){
  dataLenAux = dataLen + (int)lenDataHTTP;
  if (dataLenAux <= 0) {
     printf("error TextHandler/insertData:: dataLenAux<0 \n");
     exit(-1);
  }
  dataAux = (u_char*)malloc(dataLenAux);
  if (dataAux == NULL) {
     printf("error TextHandler/insertData:: memory failed
                    in dataAux\n");
     exit(-1);
  }
  if (data != NULL) {
     bcopy(data,dataAux,dataLen);
     dataAux = dataAux + dataLen;
  }
  bcopy(d,dataAux,(int)lenDataHTTP);
  dataAux = dataAux - dataLen;

  if (data != NULL){
    free(data);
    data = NULL;
  }
  data = dataAux;
  dataLen = dataLenAux;
}else{
  printf("error TextHandler/inserData:: lenDataHTTP is <=0 or
          d is null \nlenDataHTTP:%i\n d=%u\n",lenDataHTTP,d);
  exit(-1);
}
```

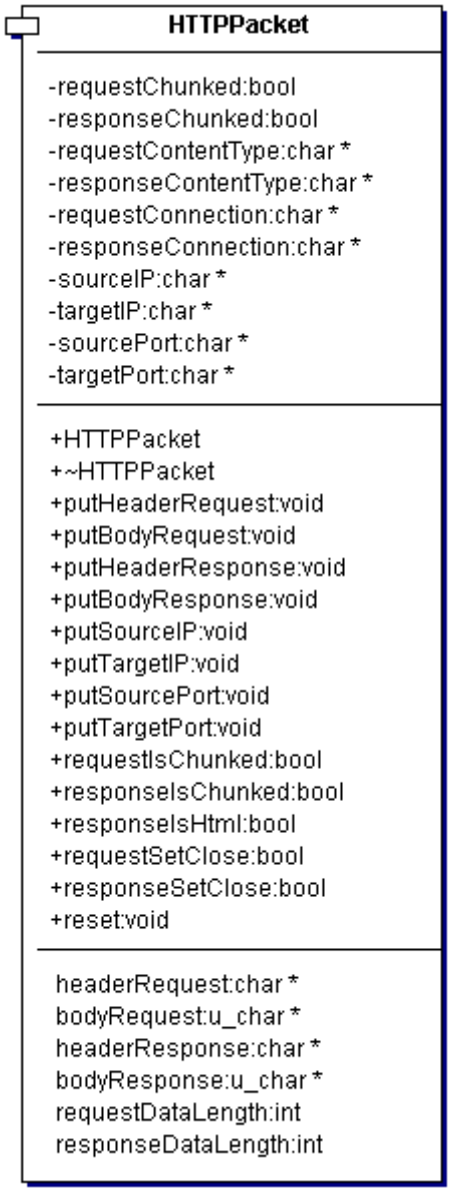
- **int getDataLength():** Método que devuelve la longitud de los datos http.

6.11 Clase HTTPPacket

- **Funcionamiento general y objetivos de la clase.**

  El objetivo principal de esta clase es almacenar una petición http y su correspondiente respuesta para enlazarla posteriormente con el conjunto de sesiones de navegación. Tanto para la petición como para la respuesta se almacena por separado la cabecera del cuerpo, permitiendo un acceso individualizado.
  Contiene además variables que almacenan valores de los headers más importantes de las cabeceras tanto de la petición como de la respuesta.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **headerRequest**: atributo que almacena la cabecera de la petición
  - **bodyResquest:** atributo que almacena el cuerpo de la petición.
  - **headerResponse**: atributo que almacena la cabecera de la respuesta.
  - **bodyResponse**: atributo que almacena el cuerpo de la respuesta.
  - **requestDataLength**: atributo que almacena la longitud del cuerpo de la petición si es que existe.
  - **responseDataLength**: atributo que almacena la longitud del cuerpo de la respuesta.
  - **requestChunked:** boolano que indica si la petición viene “troceada”.
  - **responseChunked:** boolano que indica si la respuesta viene “troceada”.

- **requestContentType**: atributo que indica el tipo del contenido del cuerpo de la petición .
- **responseContentType:** atributo que indica el tipo del contenido del cuerpo de la respuesta.
- **requestConnection:** atributo que indica si la petición incluye la cabecera Connection y almacena su valor correspondiente.
- **responseConnection:** atributo que indica si la respuesta incluye la cabecera Connection y almacena su valor correspondiente.

- **HTTPPacket():** constructor que inicializa los atributos de la clase necesarios.
- **void putHeaderRequest((char* header)):** método que fija el valor de la cabecera de la petición e inicializa los atributos requestContentType, requestChunked y demás atributos de cabeceras optativas que incluya la petición.

```
if (r!=NULL){
  int length = strlen(r);
  headerRequest = (char*)malloc(length+1);
  if (headerRequest == NULL){
     printf("error HTTPPacket/putHeaderRequest:: memory failed in
                 headerRequest\n");
     exit(-1);
  }
  bcopy(r,headerRequest,length);
  headerRequest[length] = '\0';
  char* pos = strstr(headerRequest,token);
  char* len;
  if (pos != NULL){
     pos = pos + 16;
     requestDataLength = atoi(pos);
  }
  token = "Transfer-Encoding: chunked";
  pos = strstr(headerRequest,token);
  if (pos != NULL){
    requestChunked = true;
  }
  token = "Content-Type: ";
  pos = strstr(headerRequest,token);
  if (pos != NULL){
     token = "\r\n";
     pos = pos + 14;
     char* posEnd = strstr(pos,token);
     int length = 0;
     if (posEnd != NULL) length = posEnd - pos;

     if (length > 0) {
        requestContentType = (char*) malloc(length+1);
```

```
        if (requestContentType == NULL){
           printf("error HttpPacket/putHeaderRequest:: memory
                   failed in requestContentType\n");
           exit(-1);
        }
        memcpy(requestContentType,pos,length);
        requestContentType[length] = '\0';
    }
  }
  token = "Connection: ";
  pos = strstr(headerRequest,token);
  if (pos != NULL){
    token = "\r\n";
    pos = pos + 12;
    char* posEnd = strstr(pos,token);
    int length = 0;
    if (posEnd != NULL) length = posEnd - pos;
        if (length > 0) {
           requestConnection = (char*) malloc(length+1);
           if (requestConnection == NULL){
              printf("error HttpPacket/putHeaderRequest:: memory
                     failed in requestConnection\n");
              exit(-1);
           }
           memcpy(requestConnection,pos,length);
           requestConnection[length] = '\0';
        }
  }
}else{
   printf("error HttpPacket/putHeaderRequest:: header request
           is null\n");
   fflush(stdout);
   exit(-1);
}
```

- **void putBodyRequest(u_char* body,int lenBody):** método que fija el valor del cuerpo de la petición si es que tiene.

```
if (body!=NULL){
  if (bodyRequest!=NULL) {
     printf("error HttpPacket/putBodyRequest:: bodyRequest is
              null\n");
      fflush(stdout);
      exit(-1);
  }
  bodyRequest = (u_char*)malloc(lenBody);
  if (bodyRequest == NULL){
     printf("error HttpPacket/putBodyRequest:: memory failed in
             bodyRequest\n");
     fflush(stdout);
     exit(-1);
  }
```

```
  bcopy(body,bodyRequest,lenBody);
}else{
    printf("error HttpPacket/putBodyRequest:: body is null\n");
    fflush(stdout);
    exit(-1);
}
```

- **void putHeaderResponse(char* header):** método que fija el valor de la cabecera de la respuesta e inicializa los atributos responseContentType, responseChunked y demás atributos de cabeceras optativas que incluya la petición.

```
if (r!=NULL){
  int length = strlen(r);
  headerResponse = (char*)malloc(length+1);
  if (headerResponse == NULL){
     printf("error HttpPacket/putBodyResponse:: memory failed in
             headerResponse\n");
     fflush(stdout);
     exit(-1);
  }
  bcopy(r,headerResponse,length);
  headerResponse[length] = '\0';
  char* token = "Content-Length: ";
  char* pos = strstr(headerResponse,token);
  char* len;
  if (pos != NULL){
     pos = pos + 16;
     responseDataLength = atoi(pos);
  }
  token = "Transfer-Encoding: chunked";
  pos = strstr(headerResponse,token);
  if (pos != NULL){
    responseChunked = true;
  }
  token = "Content-Type: ";
  pos = strstr(headerResponse,token);
  if (pos != NULL){
    token = "\r\n";
    pos = pos + 14;
    char* posEnd = strstr(pos,token);
    int length = 0;
    if (posEnd != NULL) length = posEnd - pos;
    if (length > 0) {
      responseContentType = (char*) malloc(length+1);
      if (responseContentType == NULL){
         printf("error HttpPacket/putBodyResponse:: memory failed
                 in responseContentType\n");
         fflush(stdout);
         exit(-1);
      }
```

```
      memcpy(responseContentType,pos,length);
      responseContentType[length] = '\0';
    }
  }
  token = "Connection: ";
  pos = strstr(headerResponse,token);
  if (pos != NULL){
    token = "\r\n";
    pos = pos + 12;
    char* posEnd = strstr(pos,token);
    int length = 0;
    if (posEnd != NULL) length = posEnd - pos;
    if (length > 0) {
       responseConnection = (char*) malloc(length+1);
       if (responseConnection == NULL){
          printf("error HttpPacket/putBodyResponse:: memory failed
                 in responseConnection\n");
          exit(-1);
       }
       memcpy(responseConnection,pos,length);
       responseConnection[length] = '\0';
    }
  }

}else{
  printf("ERROR HttpPacket/putBodyResponse:: header response is
            null\n");
}
```

- **void putBodyResponse(u_char* body,int lenBody):** método que fija el valor del cuerpo de la respuesta.

```
if(body!=NULL){
  if (bodyResponse!=NULL) {
      printf("error HttpPacket/putBodyResponse:: bodyResponse is
             not null\n");
      fflush(stdout);
      exit(-1);
  }
  bodyResponse = (u_char*)malloc(lenBody);
  if (bodyResponse == NULL){
       printf("error HttpPacket/putBodyResponse:: memory failed in
              bodyResponse\n");
       fflush(stdout);
       exit(-1);
  }
   bcopy(body,bodyResponse,lenBody);
}
```

- **char* getHeaderRequest():** devuelve el valor de la variable headerRequest.

- **u_char* getBodyRequest():** devuelve el valor de la variable bodyRequest.
- **char* getHeaderResponse():** devuelve el valor de la variable headerResponse.
- **u_char* getBodyResponse():** devuelve el valor de la variable bodyResponse.
- **int getRequestDataLength():** devuelve el valor de la variable requestDataLength.
- **int getResponseDataLength():** devuelve el valor de la variable responseDataLength.
- **bool requestIsChunked():** método que devuelve cierto si la petición viene "troceada".
- **bool responseIsChunked():** método que devuelve cierto si la respuesta viene "troceada".
- **bool responseIsHtml():** método que devuelve cierto si la respuesta es una página html.

```
if(responseContentType != NULL)
   return (strcmp(responseContentType,"text/html") == 0);
else return false;
```

- **bool requestSetClose():** método que devuelve cierto si la petición viene "troceada".

```
if (requestConnection != NULL){
    int a = strcmp(requestConnection,"close");
    return (strcmp(requestConnection,"close") == 0);
}else return false;
```

- **bool responseSetClose():** método que devuelve cierto si en la cabecera de la respuesta incluye el atributo Connection:close.

```
if (responseConnection != NULL)
    return (strcmp(responseConnection,"close") == 0);
else return false;
```

- **void reset():** método que libera la memoria reservada y prepara las estructuras necesarias para almacenar un nuevo paquete http.
- **void putTargetIP(char* r):** asigna el valor de la IP destino del paquete HTTP.

- **void putSourceIP(char* r):** asigna el valor de la IP fuente del paquete HTTP.
- **void putTargetPort(char* r):** asigna el valor del puerto destino del paquete HTTP.
- **void putSourcePort(char* r):** asigna el valor del puerto fuente del paquete HTTP.

6.11 Clase Event

- **Funcionamiento general y objetivos de la clase.**

  Esta clase sirve para gestionar los diferentes eventos de navegación. Hay que controlar cuando han llegado todas las peticiones automáticas que se necesitan para completar la carga del evento o cuando las tenemos que dejar de esperar porque se ha cumplido el tiempo de timeout.

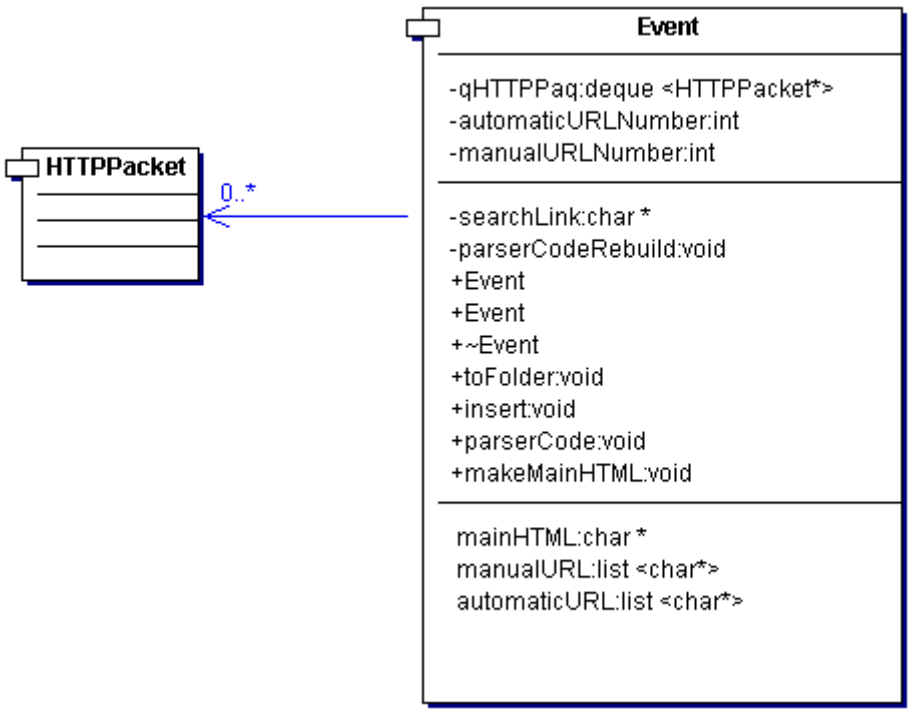
  El evento tiene el html principal que es el que desencadena la lista de peticiones automáticas que se realizarán. Este html habrá que pasearlo varias veces: para identificar las peticiones automáticas que el navegador hará para completar la visualización de la página completa y otra para modificarlo cambiando las referencias imágenes, sonidos, etc, por enlaces locales. En el parseo es necesario identificar también las peticiones manuales que aparecen en la página que va a permitir identificar la página previa a la que nos encontramos. De esta forma es posible enlazar las diferentes sesiones.

  Al realizar el parseo hay que tener en cuenta donde se encuentran las peticiones manuales y las automáticas. En el caso de las automáticas, se identifican a través de la etiqueta html src, de manera que lo que venga detrás de esta etiqueta va a ser una petición automática. En el caso de las manuales, hay más posibilidades y habría que tener en cuenta más etiquetas: href, url, action, value. Las peticiones manuales pueden aparecer como un enlace a otra página, como un botón, al enviar un formulario u otro tipo de datos, etc. Hay que tener en cuenta que no vamos a analizar las peticiones que se encuentren incluidas en código javascript, solo las que se encuentren en código html.

  En esta clase tenemos que obtener también la dirección de la página principal que es quien constituye el evento. Para ello, hay que realizar un parseo en la cabecera de la petición, incluso en posible que no sea posible encontrar la dirección de esta página, por lo que hay que tener en cuenta todos los casos.

  Para poder llevar a cabo todas estas funciones, la clase dispone de un conjunto de atributos y métodos que facilitan su realización. A pesar de que la clase esté implementado no está funcionando con el proyecto.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **manualURL:** Lista que contiene todas las peticiones manuales
  - **automaticURL:** Lista que contiene todas las peticiones automáticas
  - **qHTTPpaq:** Lista que contiene todos los HTTPPackets que se corresponden con peticiones automáticas.
  - **mainHTML:** Dirección del html. Se puede obtener mediante host o referer. Se puede hacer una combinación de ambas si no aparece la dirección completa en una de ellas.
  - **automaticURLNumber:** Número de peticiones automáticas
  - **manualURLNumber:** Número de peticiones manuales

  - **char* searchLink(char* pos, bool automatic):** Cada vez que encontramos una etiqueta que puede contener una petición se utiliza este método para extraer dicha posición. Automatic es true siempre que la etiqueta encontrada sea src. Las peticiones pueden venir entre comillas simples, dobles o sin comillas. Si viene entre comillas se sabe que el final va a ser también una comilla, pero si viene sin comillas el final de la petición puede ser un espacio en blanco o un cierre de etiqueta html >. En este caso, como no vamos a saber en que termina buscamos tanto el

espacio en blanco como el tag. Una vez que se han encontrado, se comprueba cual aparece antes, y el que se encuentre primero va a ser el final de la petición. Una vez que se ha encontrado hay que comprobar si es una petición completa o si hay que concatenarla con la dirección http principal. Si la petición comienza con “http” es una dirección completa y no es necearia la concatenación. En caso contrario, se concatena con la dirección que aparece en *mainHTML*. Una vez que está la dirección completa, se guarda en su lista correspondiente según sea una petición automática o manual. El método va a devolver la posición del final de la petición obtenida para continuar el parseo a partir de ese punto.

```
//Primero comprobamos si empieza por comillas
comS = strstr(pos,"\'");
comD = strstr(pos,"\"");
if(comS-pos == 0){ //viene entre comillas simples
   newPos = pos+1;
   hayComS = true;
}else if(comD-pos == 0){ //viene entre comillas dobles
        newPos = pos+1;
         hayComD = true;
      }else{//viene sin comillas
            newPos = pos;
      }

//comprobamos si viene una direccion completa o hay q unirla a la
  principal (http o /)
comp1 = strncmp(newPos,"http",4);
comp2 = strncmp(newPos,"/",1);
if(comp2 == 0) unionNecessary=true;
if(comp1 == 0 || comp2 == 0){
   //buscamos el final
   if(hayComS){
       posEnd1 = strstr(newPos,"\'");
       len=posEnd1-newPos;
   }else if(hayComD){
            posEnd1 = strstr(newPos,"\"");
            len=posEnd1-newPos;
        }else{
            //Buscamos espacio o > para terminar
            posEnd1 = strstr(newPos," ");
            posEnd2 = strstr(newPos,">");
            if (posEnd1 != NULL)
                {len1 = posEnd1-newPos;len= len1;}
            if (posEnd2 != NULL)
               {len2 = posEnd2-newPos;len= len2;}
            if (posEnd1 != NULL && posEnd2 != NULL){
               if(len1<len2) len = len1;
               else len = len2;
            }
         }//else
```

```
  if(unionNecessary){
    //hay que concatenar mainHTML con newPos
    link = (char*)malloc(strlen(mainHTML)+len+1);
    strcat(link,mainHTML);
    strcat(link,newPos);
  }else{
    link = (char*)malloc(len);
    link = strncpy(link,newPos,len);
  }

  //guardamos el link en la lista correspondiente segun sea
    automatico o manual
  if(automatic){
    automaticURL.push_back(link);
  }else{
    manualURL.push_back(link);
  }
}
return newPos;
```

- **Event(HTTPPacket* paq):** Se encarga de coger el código html ,que se encuentra en el paquete http en el cuerpo de la respuesta, para parsearlo e inicializa las variables del número de peticiones manuales y automáticas después de haber realizado el parseo.
- **void toFolder():** Recorre la cola de paquetes http de ese evento y los va almacenando en fichero. El nombre del fichero es el mismo con el que se hizo la petición. Se invocará desde el programa principal cuando esté ocioso si ha expirado el timeout o bien al insertar cuando el contador *automaticURLNumber* es 0.
- **void insert(HTTPPacket* paq):** El paquete que se le pasa como parámetro debe ser siempre la respuesta de una petición automática. Se inserta en la cola *qHTTPpaq*, se elimina de la lista de url automáticas, y se decrementa el contador. Si el contador ha llegado a cero, se invoca al método *toFolder()*.
- **void parserCode(char* code):** Se encarga de realizar el parseo del código HTML. Para ello, va recorriendo el código buscando los tag que pueden contener peticiones. Puesto que cada página puede contener las etiquetas de diferentes maneras , ya que el formato depende del creador de la página, es necesario buscar las etiquetas con distintos formatos, mayúsculas, minúsculas, con mayúscula la primera letra. Se buscan los tags con las distintas posibilidades, y siempre que se encuentre una de las etiquetas se llama al método *searchLink* para obtener la petición. Antes de llamar a este método hay que colocarse detrás del tag para facilitar la labor de *searchLink.* Para ello hacemos, *pos = pos + num* siendo num el

número de letras del tag que se ha encontrado mas 1 para saltarse también el = .

```
char* pos1 = strstr((char*)code,hrefMin);
char* pos2 = strstr((char*)code,hrefMay);
char* pos3 = strstr((char*)code,hrefF);
char* pos4 = strstr((char*)code,srcMin);
char* pos5 = strstr((char*)code,srcMay);
char* pos6 = strstr((char*)code,srcF);
char* pos7 = strstr((char*)code,actionMin);
char* pos8 = strstr((char*)code,actionMay);
char* pos9 = strstr((char*)code,actionF);
char* pos10 = strstr((char*)code,valueMin);
char* pos11 = strstr((char*)code,valueMay);
char* pos12 = strstr((char*)code,valueF);
char* pos13 = strstr((char*)code,urlMin);
char* pos14 = strstr((char*)code,urlMay);
char* pos15 = strstr((char*)code,urlF);

while(pos1!=NULL || pos2!=NULL || pos3!=NULL ||pos4!=NULL
       ||pos5!=NULL ||pos6!=NULL ||pos7!=NULL ||pos8!=NULL
       ||pos9!=NULL||pos10!=NULL||pos11!=NULL||pos12!=NULL
       ||pos13!=NULL ||pos14!=NULL ||pos15!=NULL){

//pos=pos+num se utiliza para posicionarse detras del nombre ya
  encontrado(href,src,...) y despues del =
//hay que pasarle al nuevo metodo si es manual o automatica con
  un bool. Automatica->src

  if(pos1!=NULL){
       pos1 = pos1+5;
       pos1 = searchLink(pos1,false);
       pos1 = strstr(pos1,hrefMin);
  }
  if(pos2!=NULL){
       pos2 = pos2+5;
       pos2 = searchLink(pos2,false);
       pos2 = strstr(pos2,hrefMay);
  }
  if(pos3!=NULL){
       pos3 = pos3+5;
       pos3 = searchLink(pos3,false);
       pos3 = strstr(pos3,hrefF);
  }
  if(pos4!=NULL){
       pos4 = pos4+4;
       pos4 = searchLink(pos4,true);
       pos4 = strstr(pos4,srcMin);
  }
  if(pos5!=NULL){
       pos5 = pos5+4;
       pos5 = searchLink(pos5,true);
       pos5 = strstr(pos5,srcMay);
```

```
  }
  if(pos6!=NULL){
        pos6 = pos6+4;
        pos6 = searchLink(pos6,true);
        pos6 = strstr(pos6,srcF);
  }

}
.
.
.
//El resto de los casos son iguales
```

- **char* getMainHTML():** Devuelve la variable que contiene la dirección html de la página.
- **list <char*> getManualURL():** Devuelve la lista que contiene las peticiones manuales.
- **list<char*> getAutomaticURL():** Devuelve la lista que contiene las peticiones automáticas.
- **void makeMainHTML(HTTPPacket* paq):** Construye la dirección de la página principal a partir de la cabecera de la petición. La dirección se va a encontrar en el GET de la petición o en el campo host que aparece en la cabecera de la petición. Es posible que sea necesario concatenar ambos campos en el caso en el que no aparezca la dirección completa.

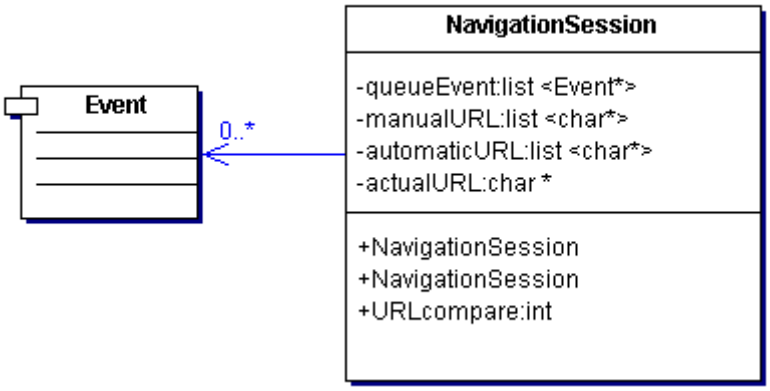
```
char* posHost;
char* posHostEnd;
char* posGet;
char* posGetEnd;
char* headerRequest=paq->getHeaderRequest();
int lenHost, lenGet;

posHost=strstr(headerRequest,"Host: ");
posGet=strstr(headerRequest,"GET ");
if (posHost==NULL || posGet==NULL) mainHTML=NULL;
else{
  //los host siempre terminan en \r\n
  posHostEnd = strstr(posHost,"\r\n");
  //los get siempre terminan en espacio
  posGetEnd = strstr(posGet," ");
  lenHost = posHostEnd-posHost;
  lenGet = posGetEnd-posGet;
  mainHTML= (char*) malloc(lenHost+lenGet+1);
  strcat(mainHTML,posHost);
  strcat(mainHTML,posGet);
}
```

6.12 Clase NavigationSession

- **Funcionamiento general y objetivos de la clase.**

El objetivo de esta clase es la de gestionar todos los eventos que pertenezcan a una misma sesión de navegación. Por sesión de navegación se entiende todo lo que es mostrado en una misma ventana del navegador. Esta clase posee una lista de eventos relacionados, ordenados según suceden a lo largo del tiempo. Es decir una lista donde está todo el itinerario que el usuario ha ido creando a lo largo de la navegación en un mismo navegador. Se poseen listas de las URLs que han sido usadas o referenciadas de alguna manera por el contenido HTTP que forman los eventos que la lista. Cuando un evento se tiene que enlazar con otro se van comparando estas listas con las listas del evento en las distintas sesiones de navegación y se decide con cual se enlaza o si es una nueva sesión de navegación.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **queueEvent:** lista de objetos de la clase Event ordenados de tal forma que un evento proviene del anterior. El último elemento de la lista es el evento más reciente, por tanto es el que está actualmente en la ventana de navegación.
  - **manualURL:** lista de char* en la cual se guardan todas las URLs “manuales”. Una URL manual es la que se ha activado directamente por el usuario, es decir, se ha ido a ese vínculo por una petición expresa independientemente de la forma en que se hace.
  - **automaticURL:** lista de char* en la cual se guardan todas las URLs “automáticas”. Una URL automática es la que se ha activado sin el

consentimiento expreso del usuario. Esto es el caso de URLs de *frames*, redirecciones de páginas, pop-ups, etc...

- **actualURL:** este atributo almacena en un *char** la URL de la página web del evento activo. Esta URL se guarda para poder ligar sesiones con una fiabilidad máxima ya que muchas paquetes tienen en su código HTTP un campo *"http referer"* el cual indica de que URL procede esa petición.

- **NavigationSession():** constructor de la clase el cual inicializa a vacía todas las listas y pone a NULL actualURL.
- **NavigationSession(HTTPPacket* paq):** constructor de la clase que toma como parámetro objeto de la clase HTTPPacket. Las listas de URLs y el atributo actualURL se inicializan con los valores del evento que se crea con el paquete HTTP.
- **int URLcompare(HTTPPacket* paq):** este método toma como parámetro un paquete HTTP del cual se extrae la URL completa de la petición. Esta se compara con las listas *manualURL* y *automaticURL*. Primero se busca si la URL del paquete HTTP está en la lista *manualURL* y luego en la lista *automaticURL*. Si está en *manualURL* el método devuelve un uno, sino devuelve un dos si está en *automaticURL* y un cero si no está en ninguna de las dos listas.

```
Event* event = new Event(paq);
//obtener la direccion url del HTTP con el q vamos a comparar
char* dir = event->getMainHTML();

/*Comparamos primero con las manuales y si encontramos
coincidencia ya no buscamos en las automaticas. En caso
contrario, buscamos en las automaticas */
typedef list <char*>::iterator itURL;
itURL itManual;
char* urlCompare;
itManual = manualURL.begin();
bool equals = false;
int comp;

while(itManual != manualURL.end() && !equals ){
       urlCompare = (*itManual);
       comp = strcmp(urlCompare,dir);
       if (comp == 0){ //coinciden
          equals = true;
       }else itManual++;
}
if (equals)
   return 1; //hay coincidencia manual
else{
```

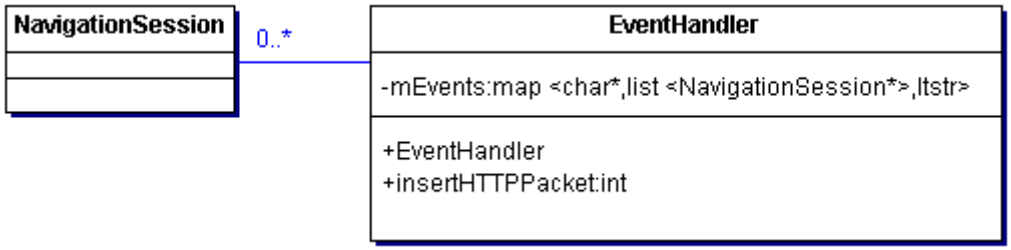
```
  itURL itAuto = automaticURL.begin();
  while ( itAuto != manualURL.end() && !equals ){
       urlCompare = (*itAuto);
       comp = strcmp(urlCompare,dir);
       if (comp == 0){ //coinciden
          equals = true;
       }else itAuto++;
  }
  if (equals)
       return 2; //hay coincidencia automatica
  else
       return 0; //no hay coincidencia
}
```

6.13 Clase EventHandler

- **Funcionamiento general y objetivos de la clase.**

El objetivo de esta clase es la de gestionar todas las sesiones de navegación que pertenecen a cada ip espiada. Para ello utilizamos una *tabla hash* con una entrada para cada ip espiada que contiene una lista de sesiones de navegación.

- **Diagrama UML.**



- **Descripción de los atributos y métodos.**

  - **mEvents:** este atributo es una *tabla hash* de listas de sesiones de navegación. La clave es la dirección ip de la máquina que se está espiando.

  - **EventHandler (int num,char** spyIps):** este es el método constructor de la clase. Tiene como parámetros el número de ip´s a espiar y una lista de char* que contiene las direcciones ip. Con la lista de ip´s se forma una estructura de directorios en los cuales se van a ir almacenando de forma ordenada los archivos que se van construyendo con el tráfico HTTP espiado. Hay una carpeta raíz */spyIps* a partir de la cual se van creando los directorios que tienen como nombre las ip´s de las máquinas que se están espiando.

```
mEvents.clear();
mkdir("/spyIps",S_IRWXU|S_IRGRP|S_IROTH);
for (int i=0;i<num;i++){
    char*dir = (char*)malloc(strlen(spyIps[i]+8));
    dir = strcpy(dir,"/spyIps/");
    strcat(dir,spyIps[i]);
    mkdir(dir,S_IRWXU|S_IRGRP|S_IROTH);
}
```

- **insertHTTPPacket(char* ip, HTTPPacket* paq):** este método toma como parámetro una dirección ip de las que se están espiando y un paquete HTTP. Lo que hace es buscar a que evento pertenece el paquete HTTP que se ha capturado de la máquina con la dirección ip que se toma como parámetro. Si se encuentra el evento al cual esté ligado se inserta el paquete y por tanto varia la sesión de navegación a la que pertenece. Si no se encuentra ningún evento al cual se puede ligar el paquete, se crea una nueva sesión de navegación y se inserta en la entrada de la *tabla hash* mEvents de la ip correspondiente. Para ver con que evento está ligado el paquete se utilizan los métodos de la clase *NavigationSession*. En cada sesión de navegación puede haber uno o varios eventos pero solo uno de ellos es el que está "activo" en un momento dado. Este evento "activo" es el que se está mostrando actualmente en el navegador de cada sesión de navegación que se está espiando. Con este evento es con el que se tiene que ver si se puede enlazar el paquete HTTP o no.

```
map <char*,list <NavigationSession*>,ltstr>::iterator iterEvents;
typedef list <NavigationSession*>::iterator iteratorNav;
typedef list <NavigationSession*> listNavigation;
typedef pair <char*,listNavigation> pairE;
iteratorNav iterNav;
iteratorNav iterManual;
iteratorNav iterAutomatic;
iteratorNav iterAutomatic2;
listNavigation listNav;
NavigationSession* nSession = new NavigationSession();
int typeURL;//tipo de url con el que coincide,0 no coincide, 1
            manual, 2 automatica
list <iteratorNav> manual;
list <iteratorNav> automatic;
manual.clear();
automatic.clear();

iterEvents = mEvents.find(ip);
if (iterEvents == mEvents.end()){
    printf("error EventHandler/insertHTTPPacket:: ip\n");
    return -1;
}else{
    listNav.clear();
    listNav=(*iterEvents).second;
    for (iterNav = listNav.begin( ); iterNav!= listNav.end( );
           iterNav++){
      nSession=*iterNav;
      typeURL=nSession->URLcompare(paq);
      if (typeURL>0){
        if (typeURL==1){
          iterManual=iterNav;
           manual.push_back(iterManual);
```

```
        }
        else{
             iterAutomatic=iterNav;
             automatic.push_back(iterAutomatic);
        }
       }
    }//End FOR
    if (manual.size()==0 && automatic.size()==0){//si es una
                                  sesion de navegacion nueva
      NavigationSession* navSession = new NavigationSession(paq);
      listNav.clear();
      listNav=(*iterEvents).second;
      listNav.push_back(navSession);
      mEvents.insert(pairE(ip,listNav));
      return 0;
    }else{
       if(automatic.size()>0){
          return 0;
       }
       else{
         return 0;
       }
}
```

6.14 Programa principal

Este el método del programa principal. Tiene dos parámetros que son el número de argumentos (*argc*) y una lista con el contenido de cada argumento (*argv*). Inicialmente se crea un objeto de la clase *Capture*, que es la que realiza básicamente la función de captura de paquetes. Se crea además un objeto de la clase *TCPSessions* para gestionar las diferentes sesiones.

Si el programa se invoca sin ningún parámetro el programa captura todo el tráfico Internet de cualquier ip.

Si se insertan parámetros de la forma indicada en las instrucciones del programa se crea el filtro adecuado.

Seguidamente se invoca el método de creación de los archivos de registro. El cual crea las estructuras de archivos necesarias para recoger la información generada por el programa.

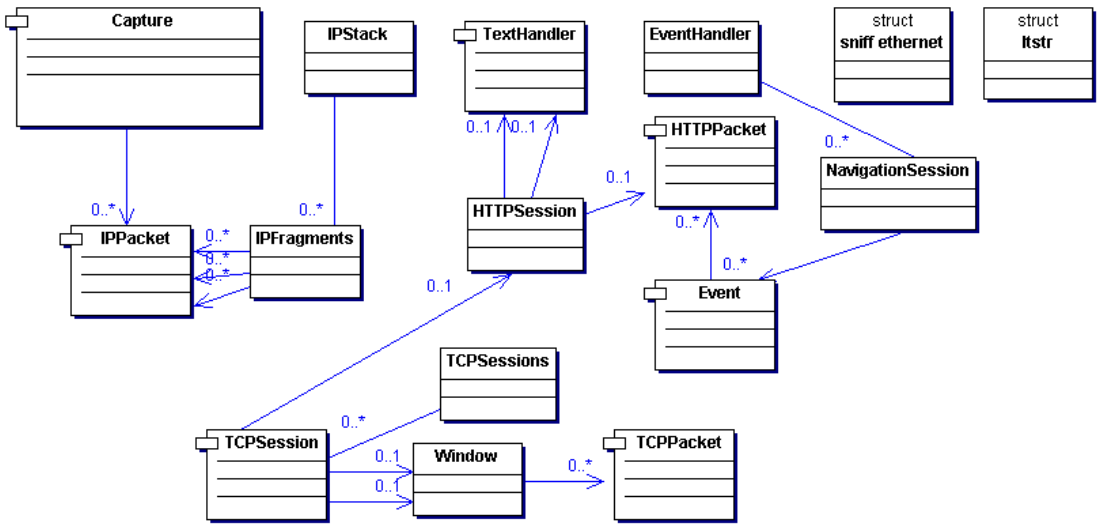
Tras el inicio de todo lo anterior, se lanza el capturador mediante la función *run()* para que capture el tráfico acorde con los valores de los parámetros. Mientras no se está capturando tráfico hay un bucle infinito que se dedica al tratamiento de los paquetes.

Este toma los paquetes de la cola de capturados si esta no es vacía, los transforma en paquetes ip, seguidamente en paquetes tcp y luego se pasan para que sean tratados por el objeto de la clase TCPSessions.

```
int main(int argc, char **argv){
      Capture * capture;
      TCPSessions* tcpSessions = new TCPSessions();

  if (argc == 1){
        capture = new Capture("port 80");
  }
  else{
      int tam=0;
      for (int i=1; i<argc;i++){
      tam = tam + strlen(argv[i]);
      }
      int tamHost=(argc-2)*5;
      int tamOr=(argc-3)*4;
      char aux[12 + tam +tamHost + tamOr];
      sprintf(aux,"port %s and (",argv[1]);
      for (int j=2; j<argc; j++){
            int tam = strlen(argv[j]);
            char aux2[5+ tam];
            sprintf(aux2,"host %s",argv[j]);
            strcat(aux,aux2);
            if (j!=argc-1) strcat(aux," or ");
            else strcat(aux,")");
      }
  capture = new Capture(aux);
  }

  makeLogFile();

  capture->run();

  while (true){
      if (!capture->isEmpty()){
            IPPacket* ipPaq = capture->popPacket();
         if (ipPaq != NULL){
            paqATratar++;
            TCPPacket* tcpPaq = new TCPPacket(ipPaq);
            tcpSessions->treatPacket(tcpPaq);
         }
      }//if
  }
}
```

7. Diseño global en UML

Aquí se muestra el diseño de todas las clases anteriormente descritas con las relaciones que hay entre ellas y el tipo de la misma.



8. Tratamiento de errores

Los errores que se dan en el programa muestran mensajes de error en la consola y algunas veces hacen que el programa aborte si es un error crítico. El formato de los errores es ***clase en la que se da el error/método en el cual se produce :: descripción del error***. Seguidamente se van a describir en profundidad los errores que se producen asi como la manera de solucionarlos

- **Clase Window:**

  - **Error Window/getHttp::memory failed:** este error se da cuando se intenta reservar memoria para la variable data del método getHttp y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **Clase HTTPPacket:**

  - **Error HTTPPacket/putHeaderRequest:: memory failed in headerRequest:** este error se da cuando se intenta reservar memoria para la variable headerRequest y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

  - **Error HttpPacket/putHeaderRequest:: memory failed in requestContentType:** este error se da cuando se intenta reservar memoria para la variable requestContentType y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

  - **Error HttpPacket/putHeaderRequest:: memory failed in requestConnection:** este error se da cuando se intenta reservar memoria

para la variable requestConnection y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **Error HttpPacket/putHeaderRequest:: header request is null:** este error se da cuando la cabecera de la petición es NULL y por tanto no es correcto el tratamiento en putHeaderRequest, ya que no es posible que una cabecera sea null. Esto puede ser debido a la captura de un paquete defectuoso o a que se ha hecho un mal tratamiento del paquete al obtener la cabecera. Compruebe el método insertRequest.

- **Error HttpPacket/putBodyRequest:: bodyRequest is null:** Este error se produce cuando el cuerpo de la petición es NULL, lo que provoca un fallo en el tratamiento de la petición. Puede ser debido a un problema en el tratamiento de la petición en un punto anterior del programa. Compruebe el método insertRequest.

- **error HttpPacket/putBodyRequest:: memory failed in bodyRequest:** este error se da cuando se intenta reservar memoria para la variable bodyRequest y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error HttpPacket/putBodyRequest:: body is null:** Este error se produce cuando el cuerpo de la petición es NULL, lo que provoca un fallo en el tratamiento de la petición. Puede ser debido a un problema en el tratamiento de la petición en un punto anterior del programa. Compruebe el método insertRequest.

- **error HttpPacket/putBodyResponse:: memory failed in headerResponse:** este error se da cuando se intenta reservar memoria para la variable headerResponse y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error HttpPacket/putBodyResponse:: memory failed in responseContentType:** este error se da cuando se intenta reservar memoria para la variable responseContentType y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **Error HttpPacket/putBodyResponse:: memory failed in responseConnection:** este error se da cuando se intenta reservar memoria para la variable responseConnection y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error HttpPacket/putBodyResponse:: header response is null:** Este error se produce cuando la cabecera de la respuesta es NULL, lo que provoca un fallo en el tratamiento de la respuesta. Puede ser debido a un problema en el tratamiento de la respuesta en un punto anterior del programa. Compruebe el método insertResponse.

- **error HttpPacket/putBodyResponse:: bodyResponse is null:** Este error se produce cuando el cuerpo de la respuesta es NULL, lo que provoca un fallo en el tratamiento de la respuesta. Puede ser debido a un problema en el tratamiento de la respuesta en un punto anterior del programa. Compruebe el método insertResponse.

- **error HttpPacket/putBodyResponse:: memory failed in bodyResponse:** este error se da cuando se intenta reservar memoria para la variable bodyResponse y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o por que se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **Clase TextHandler:**

  - **error TextHandler/getDataLength:: dataLen must be greater than headerLen:** Este error se produce cuando la longitud de la cabecera es mayor que la longitud total de los datos. Este error puede ser debido a que no identifica bien el final de la cabecera cuando tiene que encontrar \r\n\r\n. Compruebe el método getHeader.

  - **error TextHandler/insertData:: dataLenAux<0 :** dataLenAux va acumulando la longitud de los datos http , por lo que en ningún momento puede llegar a ser negativo. Para comprobar donde se produce el error compruebe los valores de dataLen y lenDataHTTP. LenDataHTTP no puede ser negativo ya que en el paso anteriror se realiza una comprobación, por lo que el error solo puede ser debido a dataLen. Compruebe el método getDataLen.

  - **error TextHandler/insertData:: memory failed in dataAux:** este error se da cuando se intenta reservar memoria para la variable dataAux y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

  - **error TextHandler/inserData:: lenDataHTTP is <=0 or d is null \nlenDataHTTP:%i\n d=%u\n ,lenDataHTTP,d :** El error se produce porque el valor de lenDataHTTP puede ser menor o igual que 0. Compruebe el método insertRequest que es donde se calcula lenDataHTTP de la clase HTTPPacket. En el caso de d compruebe el método treatPacket de la clase TCPSession.
  - **error TextHandler/containTag:: dataLen is 0 :** Si dataLen es 0 es porque todavia no se ha realizado el cálculo de este valor o porque se ha realizado incorrectamente. Compruebe el método getDataLen.

  - **error TextHandler/finishIn:: out of memory :** Se está intentando buscar un token en un char* fuera de la zona reservada de memoria. Compruebe que el valor obtenido en la búsqueda con la función *strstr* no se encuentra fuera de la zona de memoria reservada.

- **error TextHandler/finishIn:: dataLen is 0 :** Si dataLen es 0 es porque todavia no se ha realizado el cálculo de este valor o porque se ha realizado incorrectamente. Compruebe el método getDataLen.

- **error TextHandler/getHeader:: getHeader out of memory :** Se está intentando buscar un token en un char* fuera de la zona reservada de memoria. Compruebe que el valor obtenido en la búsqueda con la función *strstr* no se encuentra fuera de la zona de memoria reservada.

- **error TextHandler/getHeader:: headerLen > dataLen :** Este error puede ser debido a un mal cálculo de los valores de dataLen y headerLen. Anteriormente se comprueba que dataLen es distinto de cero por lo que el error no es debido a que no se ha calculado este valor. Compruebe el método getDataLen. Para comprobar que no hay ningún error en el valor de headerLen, compruebe el método getHeader.

- **error TextHandler/getHeader:: memory failed in header:** este error se da cuando se intenta reservar memoria para la variable header y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TextHandler/getHeader:: dataLen is 0:** Si dataLen es 0 es porque todavia no se ha realizado el cálculo de este valor o porque se ha realizado incorrectamente. Compruebe el método getDataLen.

- **Clase HTTPSession:**

  - **error HTTPSession/obtainPacket:: httpPacket is null:** Si httpPacket es NULL no va a ser posible obtener los datos que forman parte de dicho paquete. Compruebe que el paquete ha sido creado en la constructora antes de tratarlo.

  - **error TCPSession/TCPSession:: error at getting time in TCPSession :** Este error se produce al calcular el momento en el que se ha creado la sesión a partir del paquete. Compruebe quee no hay ningún error en el uso de la función *time* ni el cálculo del valor lastActivity.

- **error TCPSession/TCPSession:: memory failed in clientIP:** este error se da cuando se intenta reservar memoria para la variable clientIP y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TCPSession/TCPSession:: memory failed in serverIP:** este error se da cuando se intenta reservar memoria para la variable serverIP y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TCPSession/TCPSession:: memory failed in clientPort:** este error se da cuando se intenta reservar memoria para la variable clientPort y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TCPSession/TCPSession:: memory failed in serverPort:** este error se da cuando se intenta reservar memoria para la variable serverPort y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TCPSession/TCPSession:: a TCPSession must be created by a synchronized TCPPacket :** Este error se produce cuando aparece una TCPSession que no ha sido creada a partir de los paquetes de sincronización correspondiente. Compruebe la corrección de los estados en el método treatConnection. También puede que se deba a ejecutar el programa cuando ya ha sido cargada una página y ya se había producido anteriormente la conexión. Para que este error no se produzca cierre todas las páginas abiertas antes de ejecutar el programa.

- **error TCPSession/treatPacket:: trying to treat a packet that isn't of this session:** Aparece un paquete que no pertenece a ninguna sesión de las que se está tratando. Esto puede deberse a que la sesión a la que pertenece ya ha sido eliminada, en cuyo caso es un paquete perdido o que los datos del paquete sean incorrectos. Compruebe que el paquete TCPPacket es correcto. Este error no es grave, por lo que puede continuar la ejecución del programa.

- **error TCPSession/treatPacket:: trying to treat a data packet before making a connection:** Antes de tratar un paquete de una sesión debe haberse realizado el proceso de conexión de la sesión a la que pertenece el paquete. Compruebe la sesión a la que pertenece el paquete en función de sus ip y puertos y que el método treatConnection es correcto.

- **error TCPSessions/getKey1:: memory failed in key:** este error se da cuando se intenta reservar memoria para la variable key y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

- **error TCPSessions/getKey2:: memory failed in key:** este error se da cuando se intenta reservar memoria para la variable key y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

Clase EventHandler:

- **error EventHandler/insertHTTPPacket:: ip not found:** el error de produce al buscar la ip en mEvents al insertar paquetes http. Compruebe que las insercciones se realizan correctamente en este mismo método y que las sesiones de navegación se tratan correctamente.

- **Clase Capture:**

  - **error Capture/Capture:: memory failed in captureMode :** este error se da cuando se intenta reservar memoria para la variable captureMode y hay algún problema. La razón por la que falla puede ser porque no exista memoria suficiente, en cuyo caso es necesario revisar el programa para poder liberar memoria, o porque se está intentando reservar una cantidad de memoria negativa. Si se debe a esto hay que revisar el valor de esta variable antes de la reserva.

  - **error threadHandler::** Error producido en el tratamiento que se realiza con pcap. Se va a mostrar el error que se ha producido.

  - **error pcap_open_live():**Error producido en el tratamiento que se realiza con pcap. Se va a mostrar el error que se ha producido.

- **Main:**

Errores de creación de ficheros: Compruebe que la sintaxis es correcta y que las opciones utilizadas para la creación de dicho fichero es válida.

  - **error makeLogFile:: create file logFile**
  - **error makeLogFile:: create file logFileTCPSessions**
  - **error makeLogFile:: create file logFileTCP**
  - **error makeLogFile:: create file logFileRST**
  - **error makeLogFile:: create file logFileCodeType**
  - **error makeLogFile:: create file logFileHTTPPackets**
  - **error makeLogFile:: create file logFileRepeatGET**
  - **error makeLogFile:: create file logFileDisconnection**
  - **error makeLogFile:: create file logFileNewSession**

9.¿Qué es un sniffer?

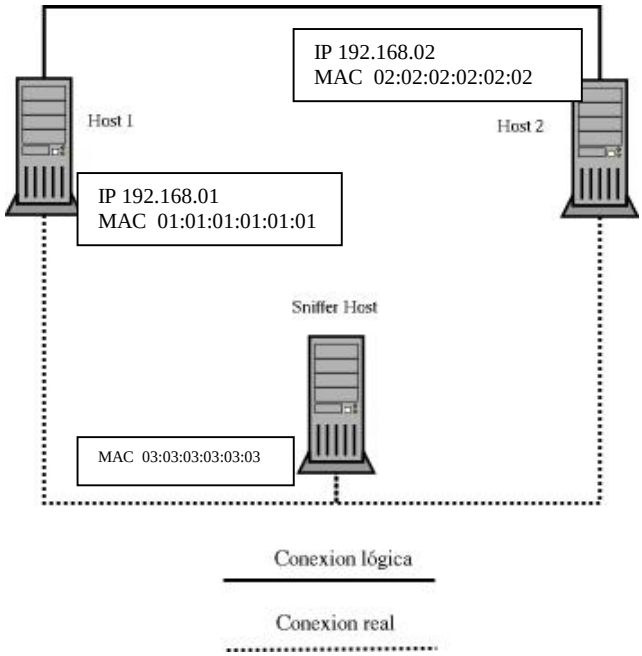
De igual manera que los circuitos de teléfono, las redes de ordenadores son canales de comunicaciones compartidos. Es simplemente demasiado costoso poner un *switch* (*hub*) para cada par de ordenadores implicados en la comunicación. El compartir significa que las computadoras pueden recibir la información que fue enviada a otras máquinas. Al capturar la información que pasa la red se llama el *sniffing*.

Normalmente la manera de conectar varios ordenadores es mediante Ethernet. El protocolo de Ethernet trabaja enviando la información del paquete a todos los *hosts* en el mismo circuito. La cabecera del paquete contiene la dirección apropiada de la máquina destino. Solamente la máquina con la dirección que va en la cabecera se supone que va a aceptar el paquete. Una máquina que está aceptando todos los paquetes, sin importar lo que ponga en la cabecera del paquete, se dice que está en modo promiscuo.

Debido a que en un ambiente normal del establecimiento de una red, la cuenta y la información de la contraseña se pasa a lo largo de Ethernet en *texto claro*, no es complicado para un atacante una vez que obtengan el *root*, poner una máquina en modo promiscuo (*sniffing*). Esto compromete todas las máquinas en la red.

Otra forma de monitorear el tráfico de una red ethernet es utilizar la técnica llamada "*arp-spoofing*". Este método no pone la interfaz de red en modo promiscuo. Esto no es necesario porque los paquetes son para nosotros y el switch enrutará los paquetes hacia nosotros.

El método consiste en "envenenar" la cache arp de las dos máquinas que queremos monitorear. Una vez que las caches estén envenenadas, los dos hosts comenzarán la comunicación, pero los paquetes serán para nosotros, los monitorearemos y los enrutaremos de nuevo al host apropiado. De esta forma la comunicación es transparente para los dos hosts. El esquema de la comunicación es sencillo:



Desde nuestra máquina enviaremos paquetes de tipo arp-reply falsos a las dos host que queremos monitorear. En estos reply's debemos de decirle al host 1 que la dirección ethernet del segundo host es la nuestra, quedando esta información almacenada en su cache arp. Este equipo enviará ahora los paquetes al host 2 pero con nuestra dirección MAC. Los paquetes ya son nuestros. El switch se encargará de hacernos llegar los datos.

Como ejemplo en el esquema anterior enviamos un flujo constante de arp-reply (para evitar que la cache arp de las maquinas se refresque con la información verdadera) al host 1 y host 2 con los siguientes datos:

HOST 1: arp-reply informando que 192.168.0.2 tiene dirección MAC 03:03:03:03:03:03

HOST 2 : arp-reply informando que 192.168.0.1 tiene dirección MAC 03:03:03:03:03:03

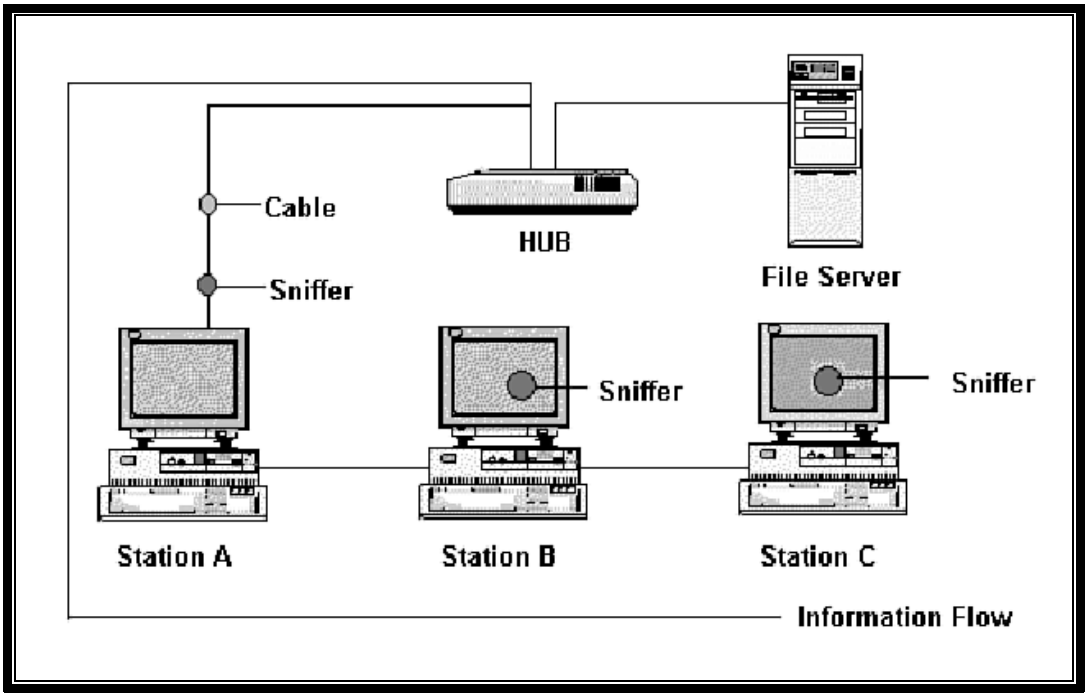
De esta forma estamos "envenenando" las cache arp. A partir de ahora lo paquetes que se envíen entre ambas nos llegarán a nosotros, pero para que ambos hosts no noten nada extraño, deberemos de hacer llegar los paquetes a su destino final. Para ello deberemos de tratar los paquetes que recibamos en función del host de origen:

Paquetes procedentes de HOST 1 ----------> reenviar a 02:02:02:02:02:02

Paquetes procedentes de HOST 2 ----------> reenviar a 01:01:01:01:01:01

De esta forma la comunicación entre ambos no se ve interrumpida, y podemos observar todo el tráfico entre ellos. Solo tendremos que utilizar un sniffer para poder capturar y filtrar el tráfico entre ambos.

Las comunicaciones entre ordenadores consisten en datos binarios aparentemente al azar. Por lo tanto, los sniffers vienen con una característica conocida como el "análisis del protocolo", que les permite "descifrar" el tráfico del ordenador y hacer que tengan sentido todos esos datos binarios capturados de los paquetes.



Esquema de un sniffer en una red ethernet

9.1 ¿Para qué se usa un sniffer?

Los programas de sniffers han estado ejecutándose por la red durante mucho tiempo en dos formas. Los programas comerciales de sniffers se usan a menudo para ayudar en el mantenimiento de las redes. Mientras que sniffers “underground” son usados por los crackers para introducirse en los ordenadores ajenos. Algunos usos típicos de estos programas son:

- Captura de passwords y logins que están en texto plano (sin encriptar) desde la red.
- Conversión de datos a un formato comprensible por los humanos
- Análisis de errores para descubrir problemas en la red.
- Análisis de rendimiento para descubrir posibles cuellos de botella en la red.
- Detección de intrusos en la red para hackers/crackers potenciales

Un sniffer más que una herramienta de ataque en manos de un administrador de red puede ser una valiosa arma para la auditoria de seguridad en la red. Puesto que el acceso a la red externa debe estar limitado a un único punto. Un sniffer puede ser la herramienta ideal para verificar como se está comportando la red.

9.2 ¿Cómo se logra el monitoreo programando?

La respuesta a esta pregunta puede ser bastante larga, pero en resumidas cuentas es posible principalmente por medio de PERL o C. Los archivos que se utilizan para la construcción de este tipo de herramientas son las librerías: linux/if.h, linux/if_ether.h, linux/ip.h, linux/socket.h, linux/tcp.h, linux/in.h, netinet/in.h, signal.h, stido.h, sys/socket.h, sys/time.h, sys/types.h. Como se puede ver en el listado anterior este sistema usa para la creación del sniffer las fuentes del sistema operativo. Para obtener más información sobre dichas fuentes puedes dirigirse a http://lxr.linux.no/.

9.3 Módulos de un sniffer

1. El hardware
La mayoría de los productos trabajan con las tarjetas de red estándar, aunque algunos requieren un hardware especial. Si se usa algún tipo de hardware especial, se puede analizar fallos como errores CRC, problemas de voltaje, programas de cable, etc.

2. Driver de captura
Ésta es la parte más importante. Captura el tráfico de la red desde el cable, lo filtra según se desee y luego almacena los datos en el buffer.

3. Buffer
Una vez que los paquetes son capturados desde la red, se almacenan en un buffer. Hay dos modos de captura distintos: captura hasta que el buffer se llene o usar el buffer como un “round robin” donde los datos más recientes reemplazan a los más antiguos.

4.Decodificar
Esta opción muestra el contenido del tráfico de la red con un texto descriptivo para que el analista sepa qué está pasando.

9.4 Sniffers para auditoría

El uso en auditoría de un sniffer es básicamente como paso de información aunque existen múltiples herramientas mucho más complejas, los sniffers son un buen

comienzo para controlar el tipo de conexiones que se realizan en el sistema. Teniendo en cuenta que esta es la forma básica que utilizan la mayoría de herramientas de seguridad, existen dos tipos de estas herramientas de host y de red, los sniffer de host se caracterizan por monitorear todo el tráfico que pasa por el computador donde están instalados. Mientras que los de Red se instalan en el punto de acceso de la red y verifican todo el tráfico que pasa dentro y hacia fuera de la red.

Cuando se instale un sniffer deberá llevar control de la información por medios automáticos ya que por el volumen que estar manejando es humanamente imposible

10. El formato de una aplicación pcap

Lo primero que hay que comprender es el funcionamiento general de un sniffer pcap. La estructura del código es la siguiente:

1. Se comienza por determinar el interfaz de red en el que queremos realizar el sniffer. En linux puede ser eht0, en BSD puede ser xl1, etc. Podemos definir este interfaz a través de un string, o podemos proporcionarle en nombre del interfaz con el que queremos trabajar.
2. Inicialización del pcap. Aquí es donde normalmente se dice al pcap el interfaz que queremos espiar. También es posible espiar varios interfaces al mismo tiempo. Para diferenciar entre estos interfaces lo que hacemos es utilizar ficheros manejadores que permiten diferenciar cada una de las sesiones con la que vamos a trabajar.
3. Podemos especificar el tráfico que vamos a querer espiar a través de un conjunto de reglas que aplicamos (solo paquetes TCP/IP, solo paquetes que van al puerto 23, etc). La regla la escribimos como un string y es convertida a un formato que el pcap pueda leer
4. Por último, ya podemos lanzar el pcap a ejecución. En este estado, el pcap espera hasta recibir alguno de los paquetes que hemos especificado. Cada vez que se obtiene un paquete se llama a una función que hemos definido. Esta función puede hacer cualquier cosa que se especifique para el tratamiento del paquete, como puede ser obtener la información del paquete e imprimirla, salvar el paquete en un fichero u otra cosa.
5. Después de finalizar el proceso del sniffer, ya podemos cerrar nuestra sesión.

Este proceso es muy simple. Cinco pasos en total de los cuales uno es opcional (el paso 3). Vamos a ver cada uno de los pasos con más detenimiento:

10.1 Especificación del interfaz

Existen dos técnicas para fijar el interfaz que deseamos espiar:

1. El usuario especifica el interfaz a través del primer argumento del programa. El string “dev” guarda el nombre del interfaz que queremos espiar en un formato que el pcap pueda comprender (asumiendo, por supuesto, que el usuario da un interfaz real).

```
#include <stdio.h>
#include <pcap.h>
int main(int argc, char *argv[])
```

```
{
    char *dev = argv[1];
    printf("Device: %s\n", dev);
    return(0);
}
```

2. La siguiente técnica es igual de simple. En este caso el pcap obtiene el interfaz a través de la función pcap_lookupdev(). Esta función devolverá eth0 si solo tenemos una tarjeta ethernet instalada en el ordenador. Si el comando falla, se mostrará un mensaje con la descripción del error. En este caso, si pcap_lookupdev() falla, mostrará el mensaje de error en errbuf.

```
#include <stdio.h>
#include <pcap.h>
int main()
{
    char *dev, errbuf[PCAP_ERRBUF_SIZE];
    dev = pcap_lookupdev(errbuf);
    printf("Device: %s\n", dev);
    return(0);
}
```

10.2 Abrir el interfaz para capturar trafico

La manera de crear una sesión sniffer es simple. Para ello se usa la función pcap_open_live(). La cabecera de esta función tiene la siguiente forma:

```
pcap_t *pcap_open_live(char *device, int snaplen, int promisc,
int to_ms, char *ebuf)
```

El primer argumento es el interfaz que hemos especificado en la sección previa. snaplen es un entero que define el número máximo de bytes que el pcap puede capturar. promisc, cuando está a true, lleva al interfaz a modo promiscuo (sin embago, incluso si está a false, es posible que haya casos específicos en los que el interfaz esté en modo promiscuo). to_ms es el tiempo que el pcap va a estar espiando en milisegundos (el valor 0 quiere decir que va a estar espiando hasta que ocurra un error; -1 captura indefinidamente). Por último, ebuf es un string que va a contener un mensaje de error.
La función va a devolver nuestra sesión manejadora.

El funcionamiento de esta función dentro del código será de la siguiente forma:

```
#include <pcap.h>
 ...
pcap_t *handle;
handle = pcap_open_live(somedev, BUFSIZ, 1, 0, errbuf);
```

Este fragmento de código abre el interfaz que se encuentra en somedev, permite leer el número de bytes especificado en BUFSIZ. Se pone el interfaz en modo promiscuo y se captura hasta que ocurra un error, y si se produce un error, muestra el mensaje que se encuentra en errbuf.

Modo promiscuo y modo no promiscuo: las dos técnicas tienen estilos muy diferentes. En el modo estándar, no promiscuo, un host va a capturar solamente el tráfico que le llega de manera directa. En el modo promiscuo, por otro lado, espía todo el tráfico de la red. Es obvia la ventaja, ya que este último modo proporciona más paquetes para capturar, los cuales pueden ser de utilidad o no según la razón por la que se esté espiando la red. Sin embargo, hay un lado negativo. Por un lado, el modo promiscuo es detectable; un host puede comprobar si otro host está funcionando en modo promiscuo . Segundo, sólo puede trabajar en entornos “non-switched” como un hub. Tercero, el elevado tráfico de la red, lo que hace que el host consuma demasiados recursos del sistema.

10.3 Filtrar el tráfico

La mayoría de las veces solo vamos a estar interesados en capturar un tráfico específico. Por ejemplo, podemos querer capturar el tráfico que viene del puerto 23 o el tráfico DNS, ya que raras veces se va a querer capturar todo el tráfico de la red. Para ello se utilizan las funciones pcap_compile() y pcap_setfilter().

El proceso es simple. Después de haber llamado a pcap_open_live() y tener una sesión de trabajo ya abierta, podemos aplicar el filtro. Hay dos razones por las que no se usan las instrucciones if/else if. La primera es porque el filtro pcap es mucho más eficiente, porque funciona directamente con el filtro BPF, ya que a través de este filtro se eliminan numerosos pasos. La segunda, es porque es mucho más fácil.

Antes de aplicar el filtro, tenemos que compilarlo. La expresión que compone el filtro se almacena en un string (char array).
Para compilar el programa se llama a pcap_compile() definida de la siguiente forma:

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str,
int optimize, bpf_u_int32 netmask)
```

El primer argumento es el manejador de nuestra sesión. El siguiente es una referencia al lugar donde se cargará la versión compilada del filtro. str es la expresión del filtro, en formato de un string regular. El siguiente es un entero que decide si la expresión debe ser optimizada o no (0 es false, y 1 es true). Por último, se especifica la máscara que se va a aplicar a la red para realizar el filtrado. La función devuelve -1 si falla.

Después de que la expresión has sido compilada, se aplica. Para ello se utiliza la función pcap_setfilter() que tiene el siguiente formato:

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

El primer argumento es nuestra sesión manejadora y el segundo es una referencia a la versión compilada de la expresión (presumiblemente la misma variable que el segundo argumento que aparece en pcap_compile()).

Un fragmento de código donde se realiza este proceso es el siguiente:

```
#include <pcap.h>
  ...
pcap_t *handle;                              /* Session handle */
char dev[] = "rl0";                        /* Device to sniff on */
char errbuf[PCAP_ERRBUF_SIZE];                 /* Error string */
struct bpf_program filter;    /* The compiled filter expression */
char filter_app[] = "port 23"          /* The filter expression */
bpf_u_int32 mask;         /* The netmask of our sniffing device */
bpf_u_int32 net;               /* The IP of our sniffing device */

pcap_lookupnet(dev, &net, &mask, errbuf);
handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
pcap_compile(handle, &filter, filter_app, 0, net);
pcap_setfilter(handle, &filter);
```

Este programa prepara al sniffer para capturar todo el tráfico que viene desde el puerto 23.

pcap_lookupnet es una función que dado el nombre de un dispositivo, devuelve su IP y máscara de red. Esto es esencial porque necesitamos conocer la máscara de red para poder aplicar el filtro.

10.4 Funcionamiento del sniffer

El funcionamiento del sniffer se basa en la captura de paquetes. Hay dos técnicas principales para esta captura. Podemos capturar un paquete cada vez o definir un bucle

que espera hasta que ha capturado n paquetes. Para la captura de un único paquete se utiliza la función pcap_next():

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

El primer argumento es la sesión manejadora. El segundo argumento es un puntero a la estructura que contiene la información general sobre el paquete, especificando el momento en el que fue capturado, la longitud del paquete y la longitud del fragmento (en caso de que esté fragmentado). pcap_next() devuelve un puntero a u_char que es el paquete descrito por esta estructura.

Demostración del uso de pcap_next():

```
#include <pcap.h>
#include <stdio.h>
int main()
{
  pcap_t *handle;                          /* Session handle */
  char *dev;                         /* The device to sniff on */
  char errbuf[PCAP_ERRBUF_SIZE];             /* Error string */
  struct bpf_program filter;           /* The compiled filter */
  char filter_app[] = "port 23";     /* The filter expression */
  bpf_u_int32 mask;                           /* Our netmask */
  bpf_u_int32 net;                                 /* Our IP */
  struct pcap_pkthdr header; /* The header that pcap gives us */
  const u_char *packet;                  /* The actual packet */
  /* Define the device */
  dev = pcap_lookupdev(errbuf);
  /* Find the properties for the device */
  pcap_lookupnet(dev, &net, &mask, errbuf);
  /* Open the session in promiscuous mode */
  handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
  /* Compile and apply the filter */
  pcap_compile(handle, &filter, filter_app, 0, net);
  pcap_setfilter(handle, &filter);
  /* Grab a packet */
  packet = pcap_next(handle, &header);
  /* Print its length */
  printf("Jacked a packet with length of [%d]\n", header.len);
  /* And close the session */
  pcap_close(handle);
  return(0);
}
```

Esta aplicación espía a un dispositivo devuelto por la llamada pcap_lookupdev() a través del modo promiscuo. Encuentra el primer paquete que llega al puerto 23 y le muestra al usuario el tamaño del paquete.

La otra técnica es más complicada pero más útil. Actualmente hay algunos sniffers que usan pcap_next(), pero la mayoría usan pcap_loop() o pcap_dispatch(). Estas dos funciones utilizan la función callback.

La idea de las funciones callback es muy simple. Supongamos un programa que está esperando a que ocurra un evento, el programa está esperando a que un usuario presione una tecla. Cada vez que la presiona, se llama a una función que indica lo que hay que hacer. Esta función que se utiliza es una callback. Cada vez que el usuario presiona una tecla, el programa llamará a la función callback. Las dos funciones que se usan para definir callback en un sniffer son pcap_loop() y pcap_dispatch(). Se llama a estas funciones cada vez que se captura un paquete y cumple con los requerimientos del filtro (en caso de que exista un filtro. Sino, todos los paquetes se envian a la función callback)

El formato de la función pcap_loop() es el siguiente:

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

El primer argumento es nuestra sesión manejadora. El siguiente es un entero que dice al pcap_loop cuantos paquetes debe tratar antes de retornar (un valor negativo significa que debe capturar paquetes hasta que ocurra un error). El tercer argumento es el nombre de la función callback. El último argumento es útil en algunas funciones pero en la mayoría de los casos es NULL.
pcap_dispatch() se usa de manera similar. La única diferencia está en como manejan los timeouts, ya que pcap_loop() ignora los timeouts mientras que pcap_dispatch() no.

El formato que se va a usar para la función callback de manera que luego pcap_loop() pueda usarlo sin problemas es:

```
void got_packet(u_char *args, const struct pcap_pkthdr *header, const
u_char *packet);
```

Esta función devuelve un void ya que pcap_loop() no sabe como manejar el valor de retorno. El primer argumento se corresponde con el último argumento de pcap_loop(), independientemente del valor que tenga. El segundo argumento es la cabecera pcap, la cual contiene información que indica cuando se capturó el paquete, su longitud,etc. La estructura pcap_pkthdr se define de la siguiente forma:

```
struct pcap_pkthdr {
    struct timeval ts;  /* time stamp */
    bpf_u_int32 caplen; /* length of portion present */
    bpf_u_int32 len;    /* length this packet (off wire) */
};
```

El último argumento es el más interesante de todos. Es otro puntero a u_char y contiene el paquete entero, capturado por el pcap_loop().

Un paquete contiene muchos atributos, por lo que no es realmente un string sino una colección de estructuras (por ejemplo, un paquete TCP/IP debería tener una cabecera Ethernet, una cabecera IP, una cabecera TCP y por último, el contenido del paquete). Para poder hacer uso de estas estructuras es necesario realizar un casting.

Primero, tenemos que tener las estructuras actuales para poder realizar un casting. La definición de estas estructuras es la siguiente:

```
/*Ethernet header*/
struct sniff_ethernet
{
    u_char ether_dhost[ETHER_ADDR_LEN]; /* Destination host address */
    u_char ether_shost[ETHER_ADDR_LEN];      /* Source host address */
    u_short ether_type;                       /* IP? ARP? RARP? etc */
};


/* IP header */
struct sniff_ip {
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int ip_hl:4,                            /* header length */
    ip_v:4;                                        /* version */
    #if BYTE_ORDER == BIG_ENDIAN
    u_int ip_v:4,                                  /* version */
    ip_hl:4;                                  /* header length */
    #endif
    #endif                                      /* not _IP_VHL */
    u_char ip_tos;                           /* type of service */
    u_short ip_len;                             /* total length */
    u_short ip_id;                             /* identification */
    u_short ip_off;                  /* fragment offset field */
    #define IP_RF 0x8000            /* reserved fragment flag */
    #define IP_DF 0x4000                /* dont fragment flag */
    #define IP_MF 0x2000               /* more fragments flag */
    #define IP_OFFMASK 0x1fff    /* mask for fragmenting bits */
    u_char ip_ttl;                              /* time to live */
    u_char ip_p;                                    /* protocol */
```

```
    u_short ip_sum;                              /* checksum */
    struct in_addr ip_src,ip_dst;  /* source and dest address */
};


/* TCP header */
struct sniff_tcp {
    u_short th_sport;           /* source port */
    u_short th_dport;      /* destination port */
    tcp_seq th_seq;         /* sequence number */
    tcp_seq th_ack;  /* acknowledgement number */
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int th_x2:4,                /* (unused) */
    th_off:4;                  /* data offset */
    #endif
    #if BYTE_ORDER == BIG_ENDIAN
    u_int th_off:4,             /* data offset */
    th_x2:4;                      /* (unused) */
    #endif
    u_char th_flags;
    #define TH_FIN 0x01
    #define TH_SYN 0x02
    #define TH_RST 0x04
    #define TH_PUSH 0x08
    #define TH_ACK 0x10
    #define TH_URG 0x20
    #define TH_ECE 0x40
    #define TH_CWR 0x80
    #define TH_FLAGS
(TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
    u_short th_win;            /* window */
    u_short th_sum;          /* checksum */
    u_short th_urp;    /* urgent pointer */
};
```

Pcap usa las mismas estructuras que los paquetes, por lo que se puede crear un u_char como tipo de los paquetes y acceder a las estructuras.
Cuando se captura un paquete de la red, es necesario tratarlo para obtener las cabeceras ethernet, IP, TCP y los datos. Esto se realiza de la siguiente forma:

```
const struct sniff_ethernet *ethernet;  /* The ethernet header */
const struct sniff_ip *ip;                   /* The IP header */
const struct sniff_tcp *tcp;                /* The TCP header */
const char *payload;                        /* Packet payload */
```

Se declaran variables con el tamaño de las estructuras:

```
int size_ethernet = sizeof(struct sniff_ethernet);
int size_ip = sizeof(struct sniff_ip);
int size_tcp = sizeof(struct sniff_tcp);
```

Se realiza el casting al tipo de las estructuras que se utilizan:

```
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + size_ethernet);
tcp = (struct sniff_tcp*)(packet + size_ethernet + size_ip);
payload = (u_char *)(packet + size_ethernet + size_ip + size_tcp);
```

La manera de acceder a cada una de las estructuras es la siguiente: supongamos una estructura ethernet de tamaño 14 bytes, una estructura ip de 20 bytes y una estructura tcp de 20 bytes. El puntero a u_char lo que realmente contiene es una dirección de memoria. Para simplificar vamos a suponer que el puntero tiene un valor X. De esta forma, si las estructuras están colocadas una detrás de otra, la primera de ellas(ethernet) se encuentra localizada en la dirección de memoria X, por lo que es sencillo encontrar la dirección del resto de las estructuras:

Variable	Localización (en bytes)
ethernet	X
ip	X + 14
tcp	X + 14 + 20
datos	X + 14 + 20 + 20

Obtención de ethernet e IP:

```
const struct ip *myip;
const struct sniff_ethernet *ethernet;
const u_char* dataAux;

int size_ethernet = sizeof(struct sniff_ethernet);
int size_ip = sizeof(struct ip);

ethernet = (struct sniff_ethernet*)(packet);
myip = (struct ip*)(packet + size_ethernet);
```

Obtención TCP:

```
const struct tcphdr *mytcp;
const struct sniff_ethernet *ethernet;
int size_ethernet = sizeof(struct sniff_ethernet);
```

```
      mytcp = (struct tcphdr*)ipPacket->getData();

      dataLength=ipPacket->getDataLength()-(mytcp->doff*4);

      dataAux = (const u_char *)(ipPacket->getData() + mytcp->doff*4);
      data=(const u_char*)malloc(dataLength);
```

10.5 Aplicación de pcap en el proyecto

Comienzo de la ejecución. Se crea un hilo que es el que se va a encargar de la captura de los paquetes cuando exista tráfico en la red. Para la captura utiliza la función threadHandler que es quién se encarga del tratamiento de los paquetes mediante la aplicación del filtro, control de fallos, etc.

```
void Capture::run(){
      int error;
      pthread_t idHilo;
      error = pthread_create(&idHilo, NULL, threadHandler, this);
}
```

Lo primero que tenemos que especificar es el tráfico que queremos espiar a través de la aplicación de un filtro:

```
       capture = new Capture("port 80 and host 147.96.80.135");
```

De esta forma se indica que solo vamos a capturar los paquetes del puerto 80 y que vayan o vengan del host 147.96.80.135.

De esta forma, los paquetes qe capturemos los vamos a pasar a la función que lleva a cabo el tratamiento de los paquetes capturados, *handle_packets.*

```
void* threadHandler(void *param){
    char *dev;
    char errbuf[PCAP_ERRBUF_SIZE];
    pcap_t* descr;
    struct pcap_pkthdr hdr;
    bpf_u_int32 net;
    bpf_u_int32 mask;
    struct bpf_program filter;
    dev=pcap_lookupdev(errbuf);
```

```
    pcap_lookupnet(dev,&net,&mask,errbuf);

    if(dev == NULL)
    {
      printf("%s\n",errbuf);
    }
    printf("DEV: %s\n",dev);
    descr = pcap_open_live(dev,BUFSIZ,0,-1,errbuf);
    if(descr == NULL)
    {
        printf("pcap_open_live(): %s\n",errbuf);
        exit(1);
    }
    char* mode = ((Capture*)param)->getCaptureMode();
    pcap_compile(descr,&filter,mode,0,net);
    pcap_setfilter(descr,&filter);

    int i=pcap_loop(descr,-1,handle_packets,(u_char*)param);

}
```

A partir de esta función va a comenzar el tratamiento de los paquetes transformado dicho paquete en IPPacket.

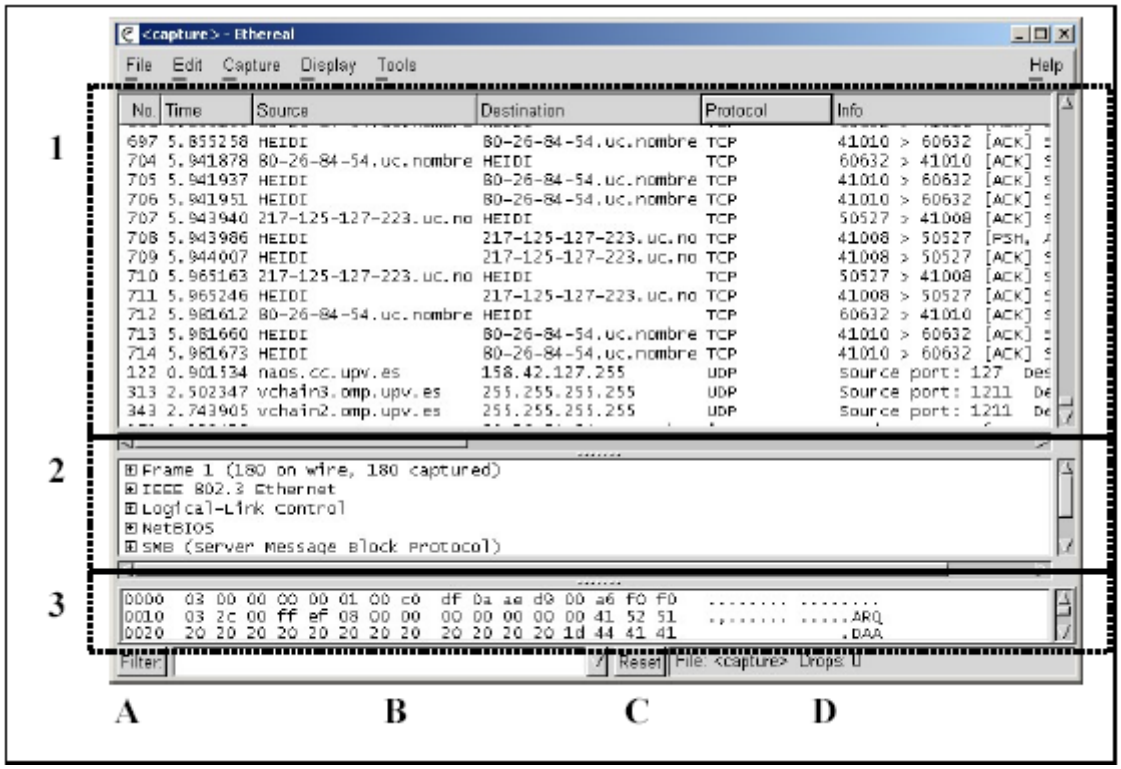
```
void handle_packets(u_char *useless,const struct pcap_pkthdr* pkthdr,
                    const u_char* packet){
  IPPacket *ipPaq = new IPPacket(packet);
  ((Capture*)useless)->pushPacket(ipPaq);

  numPackets++;
}
```

11. Utilización de ethereal

Ethereal es una aplicación que ofrece una interfaz sencilla de utilizar y que permite visualizar los contenidos de las cabeceras de los protocolos involucrados en una comunicación de una forma muy cómoda.

11.1 Interfaz y menús

Ethereal funciona en modo gráfico y está programado con la librería de controles GTK. La ventana principal de la aplicación se divide en tres partes de visualización y una zona inferior de trabajo con filtros.



Ventana principal del Ethereal

En la primera parte (1) se muestra la información más relevante de los paquetes capturados , como , por ejemplo , las direcciones IP y puertos involucrados en la comunicación. Seleccionando un paquete en esta sección podemos obtener información detallada sobre él en las otras dos secciones de la pantalla que comentaremos a continuación.

En la parte central de la ventana (2) se muestra, utilizando controles tree-view, cada uno de los campos de cada una de las cabeceras de los protocolos que ha utilizado el paquete para moverse de una máquina a la otra. Así, si hemos capturado una serie de

paquetes de, por ejemplo, una conexión telnet, podremos ver las cabeceras del protocolo TCP, del IP y de los que tengamos debajo de ellos (trama Ethernet, por ejemplo, en una red Ethernet).

La tercera parte de la ventana (3) muestra un volcado hexadecimal del contenido del paquete. Seleccionando cualquier campo en la parte central de la ventana se mostrarán en negrita los datos correspondientes del volcado hexadecimal, los datos reales que están viajando por la red.
En la barra inferior, aparecen cuatro componentes muy interesantes a la hora de hacer análisis de capturas: Creación de filtros (A), filtro actual (B), borrar filtro (C) y mensajes adicionales (D). Para obtener un mayor detalle de estos menús, se debe consultar la guía del usuario en la página Web de la aplicación.

Todas las opciones que pueden ser empleadas, son accesibles por medio de los menús, la aplicación contiene los siguientes menúsfiltros y protocolos.

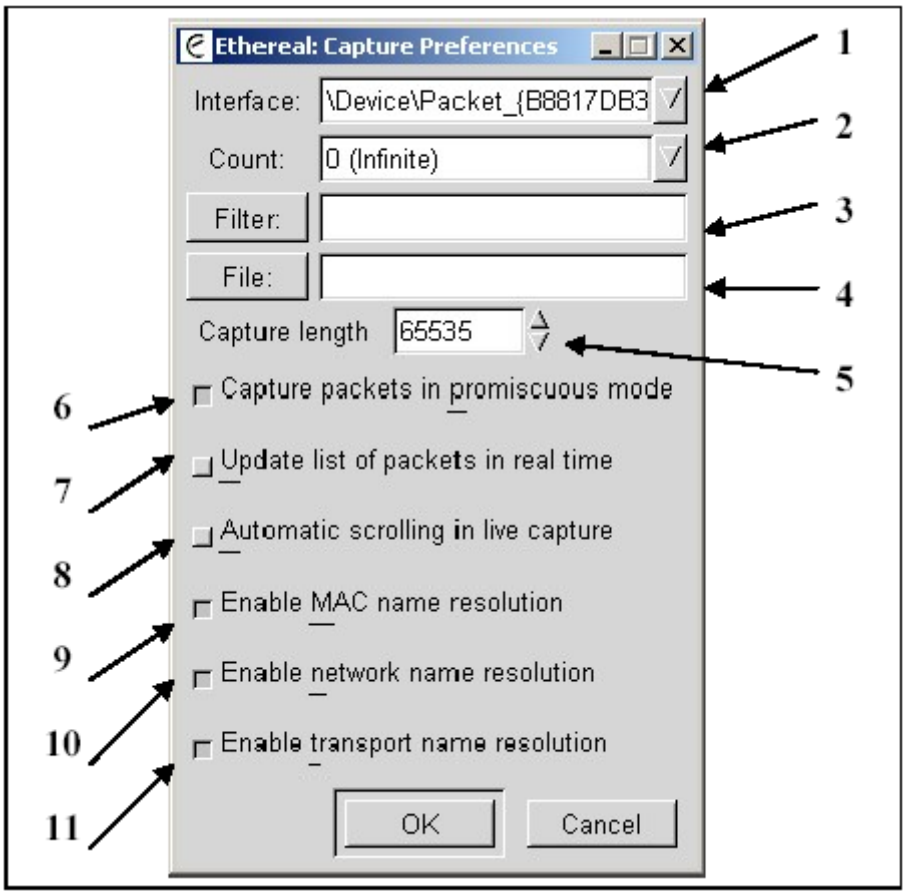
Menús de Ethereal

• **File:** Menú con los ítems para abrir, guardar ficheros de captura. Permite imprimir y salir de la aplicación.
• **Edit**: Menú para encontrar tramas concretas, ir a una trama y marcar tramas. También tiene las opciones de preferencias, de captura y visualización defiltros y protocolos.
• **Capture:** Inicia o detiene la captura de paquetes.
• **Display:** Permite cambiar las opciones de visualización, correspondencia y coloreado de marcos.
• **Tools:** Este menú contiene los pluggins instalados, seguimiento de un paquete TCP (interesante función), sumario de los paquetes capturados y la visualización de las estadísticas de protocolos empelados.
• **Ayuda:** Ayuda de la aplicación y "acerca de".

11.2 Captura de paquetes

La captura de paquetes se realiza por medio de la opción de menú capture → start. Al seleccionar la opción aparecerá la caja de diálogo con las preferencias para la captura de los paquetes.

En nuestro caso, los paquetes que nos interesa capturar son los que se generan desde o hacia la IP 147.96.80.135, y solo nos interesan los paquetes TCP/IP.



Ventana de preferencias de la captura

Las opciones que permite esta ventana son las siguientes:

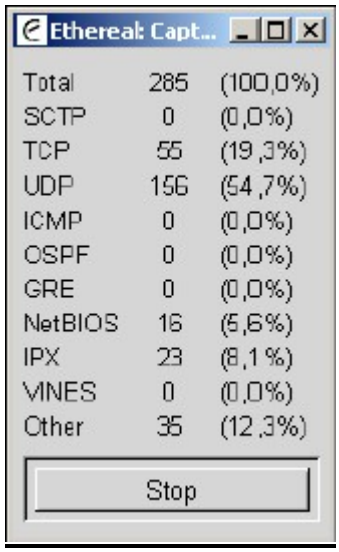
1. **Interfaz**. Es el interfaz de captura que se va a emplear para la sesión, en algunos sistemas es equivalente a las conexiones de red de las que se dispone (PPP, Red, etc…) si al seleccionar una opción no se capturan datos, se debe aque no se ha seleccionado el interfaz correcto.
2. **Cuenta**. Es el número de paquetes que deseamos que se capturen. En el caso de dejarse a 0, se capturarán todos los paquetes hasta que se detenga manualmente la sesión (o se sature el sistema).
3. **Filtro**. Determina el filtro que deseamos que se aplique a la captura. La sintaxis de los filtros se verá más adelante.
4. **Archivo**. En caso de que se desee que la captura sea volcada hacia un archivo, se puede seleccionar por medio de este cuadro de texto.
5. **Longitud**. Marca la cantidad máxima de bytes de cada trama que deseamos sean capturados.
6. **Captura de datos en modo promiscuo**. En caso de seleccionarse esta opción, se capturarán todos los datos posibles que sean alcanzables por el host monitor. Si no se selecciona1, sólo se capturarán los paquetes que entren o salgan al host monitor.

7. **Actualización de la lista en tiempo real**. Actualiza la ventana de paquetes capturados a la vez que éstos son capturados. Esta opción tiene el problema de cargar en exceso el sistema.
8. **Desplazamiento de la lista de paquetes capturados**. Permite que la ventana con el contenido de los paquetes capturados, se actualice en tiempo real.
9. **Permitir resolución de nombres MAC**. Este botón permite controlar si *Ethereal* traduce o no los primeros tres octetos de las direcciones MAC al nombre del fabricante a quien ese prefijo ha sido asignado por el IETF.
10. **Permite resolución del nombre de la red**. Este botón permite que controlar si *Ethereal* traduce o la direcciones IP a nombres del dominio del DNS. Haciendo clic en este botón, la lista de paquetes capturados tendrá información más útil, pero provocará las correspondientes peticiones de operaciones de búsqueda, que pueden influir en la captura.
11. **Permitir la resolución de nombres de transporte**. Este botón permite controlar si *Ethereal* traduce o no los números de puerto del protocolo.

En nuestro caso, solo vamos a utilizar la opción **filter** para aplicar el siguiente filtro:

port 80 and host 147.96.80.135

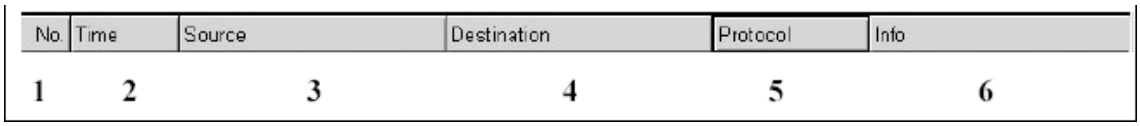
Una vez configurada la ventana con las opciones que deseemos, al presionar el botón de OK, comenzará la captura de paquetes, si se presiona CANCEL, no se capturará ningún paquete y se regresará al estado anterior. Durante la captura, aparecerá la caja de diálogo que muestra la evolución.



Evolución de la captura

11.3 Visualización de resultados

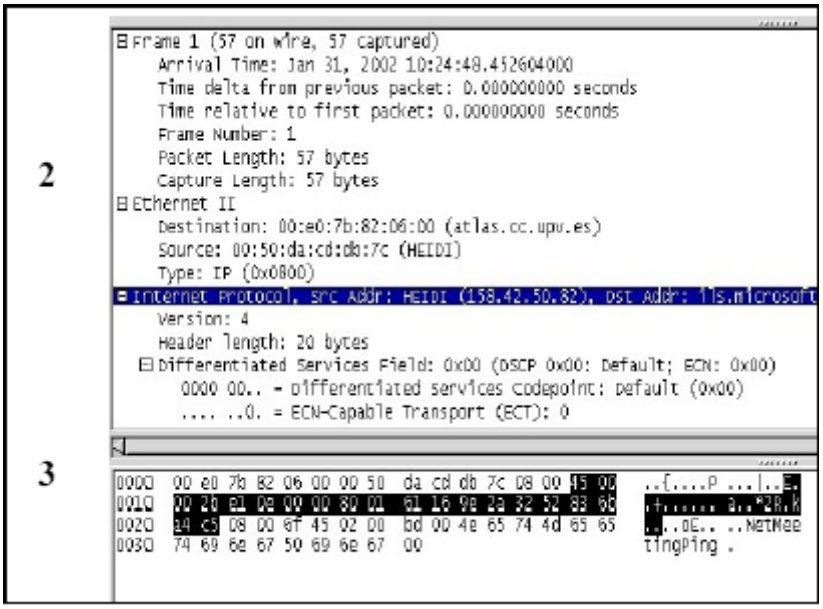
Una vez se ha capturado la sesión, toda la secuencia de estos paquetes aparecerá en los marcos de visualización. *Ethereal* proporciona una serie de herramientas que facilitan esta visualización.
La forma más directa de visualizar los resultados consiste en hacer clic en el paquete del que se desee obtener más información, dentro de la zona 1 (ver figura 1) de la ventana. Esta zona muestra los paquetes que han sido capturados. La ventana se divide en filas (una por paquete) y columnas que muestran la correspondiente información acerca de los paquetes. Por defecto las columnas que aparecen son las siguientes:



Distribución de las columnas en Ethereal

1. Número de paquete.
2. Información temporal de la captura del paquete (fecha, hora, etc.)
3. Dirección fuente.
4. Dirección destino.
5. Protocolo empleado por el paquete.
6. Información adicional.

La información de estas columnas puede ser variada por medio de la opción de menú Edit → Preferences. Se pueden añadir o quitar columnas con información adicional o variar el orden en que éstas se muestran.
Una vez se ha seleccionado un paquete en una fila, la información de éste aparece en la ventana inferior dentro de la zona 2 (ver figura 1). Allí aparece con los campos que éste tiene y, por medio de un control tree-view, la información de éstos puede ser visualizada de forma organizada.



Ventanas de información

En el panel 2. Se tiene la estructura del mensaje, al seleccionar un ítem de este panel, se visualizará su contenido en el panel 3 dentro del contenido del paquete completo como una zona sombreada. Se puede extender o contraer toda la información del marco 2 por medio de las opciones de menú Display → Collapse All y Display → Expand All, respectivamente.

Si se desea visualizar un paquete por separado, para un mejor análisis del mismo, o compararlo con otro, se puede seleccionar y por medio de la opción de menú Display→ Show packet in New Window. El paquete seleccionado aparecerá en una nueva ventana. También se puede acceder a este menú por medio del menú contextual que aparece al hacer clic con el botón derecho en el paquete seleccionado.
Otras herramientas útiles a la hora de trabajar con los paquetes visualizados son las de buscar o ir hacia un paquete con un número concreto. Esto se hace por medio de los menús Edit → Find frame, y Edit → Go to Frame. También se puede marcar un paquete por medio de la opción de menú Edit → Mark Frame (para desmarcarlo, se debe seleccionar el paquete marcado y volver a seleccionar la opción de menú Edit →Mark Frame).

11.4 Definición de filtros de captura

Ethereal permite filtrar la información acerca de los paquetes capturados, tanto en el momento de la captura de los mismos como en la visualización. *Ethereal* utiliza la misma sintaxis para la definición de filtros que la orden de Unix **tcpdump**. La descripción que se ofrece a continuación no es más que una adaptación de la información que aparece en la página man de **tcpdump**.

Un filtro de captura consiste en un conjunto de expresiones primitivas conectadas mediante los operadores lógicos **and/or** y opcionalmente precedidos por **not:**

[not] primitiva [and|or [not] primitiva ...]

tcp port 23 and host 10.0.0.5
Captura tráfico telnet (puerto 23) desde y hacia el host 10.0.0.5.
tcp port 23 and not host 10.0.0.5
Captura tráfico telnet no dirigido ni generado por el host 10.0.0.5.

Una primitiva es una de las expresiones siguientes:
[src|dst] host <host>

Permite filtrar el tráfico generado (**src**) o recibido (**dst**) por un **<host>**, indicando su dirección IP o su nombre. Si no se especifica ni **src** ni **dst**, se seleccionan todos los paquetes cuya dirección origen o destino coincide con la del computador especificado.

ether [src|dst] host <ehost>
Permite filtrar basándose en la dirección física (Ethernet). Como antes, se puede indicar **src** o **dst** para capturar sólo el tráfico saliente o entrante.

gateway host <host>
Permite filtrar paquetes que usan al <**host**> como un **gateway** (router). Es decir, paquetes cuya dirección física (origen o destino) es la del host, pero las direcciones IP (origen o destino) no corresponden al host.

[src|dst] net <net> [{mask <mask>}|{len <len>}]
Permite seleccionar paquetes basándose en las direcciones de red. Adicionalmente, se puede especificar un máscara de red o el prefijo CIDR cuando sea diferente al de la propia máquina desde donde se realiza la captura.

[tcp|udp] [src|dst] port <port>
Permite un filtrado basado en los puertos TCP y/o UDP. Las opciones **[tcp|udp][src|dst]** permiten restringir el filtrado sólo a los paquetes de un protocolo (TCP oUDP), o sólo a los que utilizan el puerto como origen o como destino.

less|greater <length>
Selecciona paquetes cuya longitud sea menor o igual (**less**) o mayor o igual (**greater)** que un valor dado **<length>.**

ip|ether proto <protocol>
Selecciona paquetes del protocolo especificado, bien al nivel Ethernet o al nivel IP.
ether|ip broadcast|multicast
Permite filtrar difusiones (**broadcast**) o multidifusiones (**multicast**) Ethernet o IP.

<expr> relop <expr>
Esta primitiva permite crear filtros complejos que seleccionan bytes o rangos de bytes en los paquetes. La primitiva se evalúa a True si se cumple la relación. **relop** es una de las siguientes: **>, <, >=, <=, =, !=,** y **expr** is una expresión aritmética compuesta por constantes enteras (expresadas en la sintaxis estándar de C), los operadores binarios **[+, -, *, /, &,** |], un operador de longitud, y funciones para acceder a los datos de un paquete. Para acceder a datos en el interior de un paquete se utiliza la sintaxis siguiente:

proto [expr : size]
proto es uno de los siguientes **ether, fddi, ip, arp, rarp, tcp, udp,** or **icmp**. El offset en bytes referido al protocolo especificado, viene dado por **expr**. **size** es opcional e indica el número de bytes del campo de interés; puede ser 1, 2 ó 4, el defecto es 1. El operador de longitud, indicado por la palabra **len**, da la longitud del paquete.

Por ejemplo:
ether[0] & 1 != 0 captura todo el tráfico multicast
ip[0] & 0xf != 5 captura todos los paquetes IP con opciones
ip[6:2] & 0x1fff = 0 captura sólo datagramas no fragmentados y el primer fragmento de un datagrama.

Esta comprobación se aplica implícitamente a todas las operaciones **tcp** y **udp** indexadas. Por ejemplo, **tcp[0]** siempre significa el primer byte de la cabecera TCP, y no el primer byte de un fragmento tcp distinto del primero.

Ejemplos:
Para capturar todos los paquetes que llegan o salen *sundown*:
host *sundown*

Para capturar el tráfico entre *helios* y, o bien, *hot*, o *ace*:
host *helios* and (*hot* or *ace*)

Para capturar todos los paquetes IP entre *ace* y cualquier host excepto *helios*:
ip host *ace* and not *helios*

Para capturar todo el tráfico entre el localhost y hosts en Berkeley:

net *ucb-ether*
Para capturar todo el tráfico ftp a través del router *snup*:
gateway *snup* and (port ftp or ftp-data)

Para capturar los paquetes de establecimiento y cierre de cada conexión TCP
(los paquetes SYN y FIN).
tcp[13] & 3 != 0

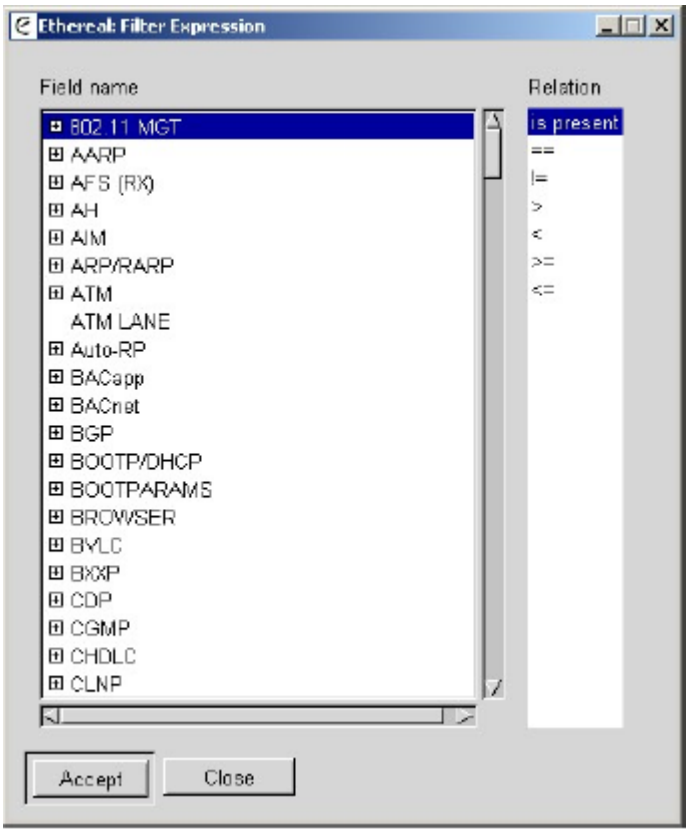
Para capturar datagramas IP mayores de 576 bytes
ip[2:2] > 576

Para capturar los datagramas IP de difusión o multidestino que no han sido enviados mediante una difusión o dirección multidestino Ethernet
ether[0] & 1 = 0 and ip[16] >= 224

Para capturar todos los paquetes ICMP, excepto las peticiones y respuestas de eco:
icmp[0] != 8 and icmp[0] != 0

11.5 Definición de filtros de visualización

Ethereal tiene dos tipos de filtros; los de captura, que siguen la nomenclatura de la instrucción UNIX tcpdump, y los de visualización que siguen una nomenclatura propia de la aplicación. Para crear un filtro de visualización, se debe crear la expresión. Ethereal dispone de una herramienta que facilita la creación de dichos filtros.



Las normas son sencillas, y para saber qué operandos pueden tener las expresiones basta con buscar el deseado en la ventana de *Ethereal Filter Expresión*. Los operadores que se pueden emplear son de comparación o los booleanos de combinación de expresiones. En las siguientes tablas se muestran estos operadores.

Campos

Los valores que pueden tomar los campos son muy variados. La explicación detallada de estos campos se puede encontrar en el apéndice A (Ethereal Display Filter Fields) del manual de uso del Ethereal. Algunos de los campos más interesantes son:

Campo		Expresión	Contenido.
tcp	srcport	tcp.srcport	Puerto del emisor (fuente).
	dstport	tcp.dstport	Puerto del receptor (destino).
ip	addr	ip.addr	Dirección IP (fuente o destino).
	dst	ip.dst	IP Destino.
	src	ip.src	IP Fuente.
	ttl	ip.ttl	Tiempo de vida.
	proto	ip.proto	Protocolo empleado.

Operadores de Comparación

Expresión	Nomenclatura C	Descripción y ejemplo
eq	==	**Igual** ip.addr==10.0.0.5
ne	!=	**No igual** ip.addr!=10.0.0.5
gt	>	**Mayor que** frame.pkt_len > 10
lt	<	**Menor que** frame.pkt_len < 128
ge	>=	**Mayor o igual que** frame.pkt_len ge 0x100
le	<=	**Menor o igual que** frame.pkt_len <= 0x20

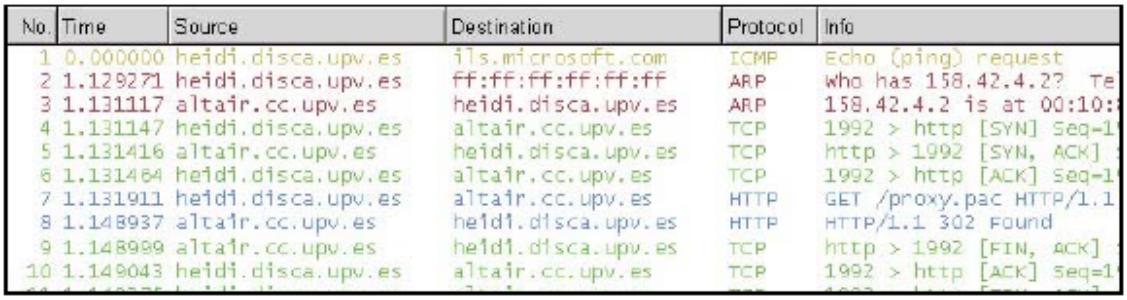
Operadores lógicos

Inglés	C	Descripción y ejemplo
and	&&	**AND Lógico** `ip.addr==10.0.0.5 and tcp.flags.fin`
or	||	**OR Lógico** `ip.addr==10.0.0.5 or ip.addr==192.1.1.1`
xor	^^	**XOR Lógico** `tr.dst[0:3] == 0.6.29 xor tr.src[0:3] == 0.6.29`
not	!	**NOT Lógico** `not llc`
[...]		**Operador de subcadena** *Ethereal* permite seleccionar subcadenas de varias formas. Después de una etiqueta se puede poner un par de corchetes [] conteniendo una lista de ítems separados por coma. `eth.src[0:3] == 00:00:83` El ejemplo superior emplea el formato n:m para especificar un rango sencillo. n es el desplazamiento inicial con respecto al comienzo del paquete y m es la longitud del intervalo.

11.6 Herramientas de análisis de la información

Aparte de las herramientas vistas en el apartado de visualización, existen otras herramientas que permiten facilitar el análisis de la misma. Una de ellas es la coloración de paquetes en función de filtros que se obtiene por medio del menú Display → Colorize Display. Para ello se debe definir un filtro que determine el criterio por el que se determinará qué paquetes se colorearán, también se escoge el color de fondo (background) y el color del texto (foreground). Una vez definidos estos datos se debe

guardar el filtro (save) y aplicarlo (apply). El resultado debe ser algo similar a lo que se observa en la figura inferior.



Coloración de paquetes

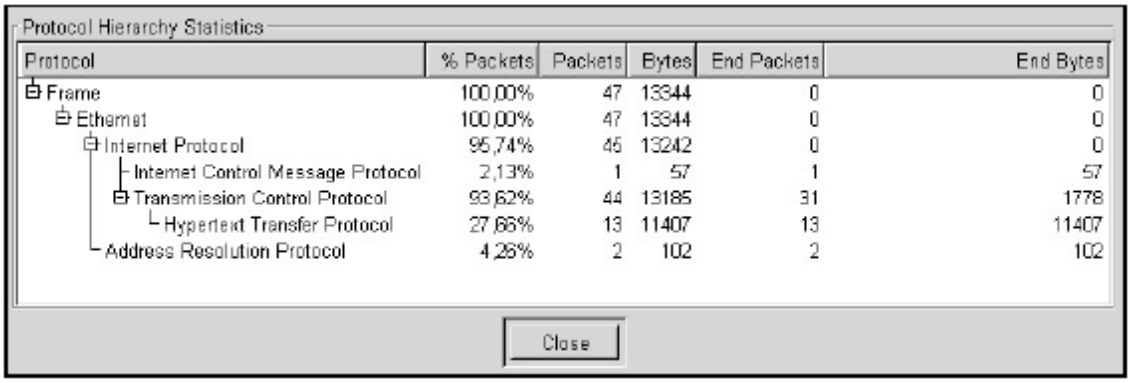
Otra opción de gran utilidad es la de seguimiento de un paquete TCP, por medio de esta opción se puede "reconstruir" la información que ha pasado por la red en una transacción TCP. Para emplear esta opción se debe seleccionar la opción de menú →Tools → Follow TCP Stream. Esta opción de menú también es contextual y aparecerá al hacer clic con el botón derecho del ratón en un paquete con el protocolo TCP. En la ventana que aparece tendremos el contenido de la secuencia de paquetes.

Ventana de seguimiento de un paquete TCP

En la ventana que se nos muestra, podemos visualizar el texto en formato hexadecimal, ASCII o EBCDIC. También se puede seleccionar el flujo a visualizar entre el entrante o el saliente de nuestra máquina. *Ethereal* obtiene el seguimiento del paquete TCP por medio de un filtro, por lo que para volver a visualizar todos los paquetes, se deberá presionar el botón de Reset (C) (ver figura 1).

Ethereal proporciona unas sencillas herramientas para un análisis básico de los paquetes que han sido capturados. Una de las opciones es el sumario de la sesión de captura (Tools→ Summary). Por medio de esta opción aparecerá una ventana con un resumen de la sesión de captura. Si se desea conocer cuál ha sido la distribución de los

protocolos y subprotocolos empleados, se dispone de la opción de menú Tools→ Protocol Hierarchy Statistics.



Resumen de protocolos

Para análisis más complejos, se deben definir los filtros adecuados. Sin embargo, *Ethereal* es un analizador de protocolos, por lo que no es una aplicación completamente indicada para evaluar de forma precisa la carga de la red, parámetros de calidad de la misma, etc.

11.7 Utilización de Ethereal en el proyecto

En un principio hemos utilizado Ethereal para familiarizarnos con los paquetes que circulan por la red y para comprobar el proceso que se realiza cada vez que se solicita una página web, ya que es lo que vamso a realizar en nuestro proyecto.

Ethereal también nos ha sido útil a la hora de realizar las pruebas del funcionamiento del proyecto:

- Número de paquetes: Hemos podido comprobar que no se pierde ningún paquete ya que cogemos el mismo número que el ethereal.
- Peticiones: Utilizamos el filtro *tcp.request == 1* de manera que solo se muestras las peticiones que se han realizado (GET). De esta forma comprobamos que no se pierde ninguna petición. En este caso también podemos comprobar el contenido de las cabeceras de las peticiones, los campos que faltan y el valor de sus campos. También podemos saber el modo en el que se va a controlar el fin de la respuesta si aparece el tamaño del paquete, ya que en el caso de que no aparezca se realiza un tratamiento especial.
- Respuestas: Del mismo modo que en el caso anterior, utilizamos el filtro *tcp.response == 1*, para obtener solo las respuestas y poder compararlas con las nuestras.

- Conexiones y desconexiones: Para comprobar que los procesos de conexión y desconexión de cada sesión se han realizado correctamente. Para ello se utilizan los siguientes filtros:
    - *tcp.port == numero* → Selecciona todos los paquetes que están relacionados con ese puerto, tanto si es origen como destino
    - *tcp.srcport == numero* → Selecciona los paquetes en los que el puerto aparece como origen
    - *tcp.dstport == numero* → Selecciona los paquetes en los que el puerto aparece como destino
- Flags: Control de los flags que aparecen en cada paquete para saber que paquetes tienen activo los flags de reset, fin, syn, etc. De esta manera podemos controlar que los distintos procesos que hay que realizar cuando aparecen paquetes con dichos flags activos se realizan correctamente.

12 Utilización de STL

Elementos que se han utilizado:

- **map** : Se utiliza como tabla hash

  #include <map> Necesario para su funcionamiento
  Declaración : map <m1,m2> nombre_map
  m1 es el tipo de la clave que se utiliza
  m2 es el tipo de los elementos que se almacenan

  map es útil como tabla hash ya que permite almacenar los datos en función a la clave, acceder a ellos, eliminarlos, etc.

  Funciones:
  - first : Acceso a la clave
  - second : Acceso al valor
  - erase(pos) : Elimina un elemento
  - clear(): elimina todos los elementos
  - count(clave) : comprueba si existe una clave en el map
  - empty() : comprueba si es vacio
  - size() : Tamaño del map
  - Insercción:
    - map[clave] = valor
    - insert(value_type(tipoClave(clave),dato))
      typedef NombreMap :: value_type value_type

      Insercción si no existe esa clave en el map

  Utilización de map en el proyecto

  - **IPStack:** Se utiliza map para ir acumulando los fragmentos de un paquete ip. Si llega un fragmento y existe una entrada con su misma clave se inserta en su posición correspondiente. En caso contrario, crea una nueva entrada para todos los fragmentos que formen parte de su mismo paquete ip.

```
map <u_short,IPFragments*> stack;
```

- **TCPSessions:** Cada entrada de la hash se encarga de controlar una sesión tcp determinada que se diferencia de las demás a través del cálculo de la clave.
  Cuando llega un paquete tcp, se comprueba si ya existe una sesión correspondiente a ese paquete. En caso afirmativo, se introduce en esa sesión, y en el otro caso, se crea una nueva sesión.
  Para el uso de map en este caso es necesario definir la función ltstr, que define la función "menor que" entre cadenas.

```
struct ltstr{
  bool operator()(const char* s1, const char* s2) const{
           return strcmp(s1, s2) < 0;
  }
};

map <char*,TCPSession*,ltstr> mSessions;
```

- **EventHandler:** Se utiliza map para manejar el conjunto de eventos de cada ip espiada, donde la clave va a a ser dicha ip. Cada entrada tendrá una lista de eventos independientes entre si.

```
  map <char*,list <NavigationSession*>,ltstr> mEvents;
```

- **deque:** Se utiliza como cola para almacenar los datos

#include < deque >
Declaración: deque <t> nombre
t es el tipo de los elementos que se van a almacenar

deque permite el almacenamiento de los datos pero no realiza ningún tipo de ordenación. Se van almacenando según el orden de llegada.

Funciones:

- push_back(elem): añade un elemento al final de la cola
- front() : devuelve una referencia al primer elemento de la cola
- pop_front() : elimina el primer elemento de la cola
- empty(): comprueba si la cola es vacia
- clear() : elimina todos los elementos de la cola
- size(): devuelve el número de elementos de la cola

Utilización de deque en el proyecto

- **Event :** Se utiliza para almacenar todos los HTTPpackets que se corresponden con peticiones automáticas.

```
deque <HTTPPacket*> qHTTPPaq;
```

- **Capture:** Los paquetes IP construidos se almacenan en esta cola para esperar su tratamiento.

```
deque <IPPacket*> queuePackets;
```

- **list** : Se utiliza para el caso en el que sea necesario mantener los elementos que se almacenan ordenados. Es necesario indicar en función de que campo se va a realizar la ordenación.

#include <list>
Declaración: list<t> nombre
t es el tipo de los elementos de la lista

Funciones:

- push_back(elem): añade un elemento al final de la lista
- front() : devuelve una referencia al primer elemento de la lista
- pop_front() : elimina el primer elemento de la lista
- empty(): comprueba si la lista es vacia
- size() : Tamaño del map
- clear() : elimina todos los elementos de la lista
- sort(): ordena los elementos de la lista en orden ascendente o en algún otro orden según se especifique

Utilización de list en el proyecto

- **IPFragments:** Se utiliza una lista para ir acumulando los paquetes Ip que son fragmentos. Se utiliza una lista porque es necesario llevar a cabo una ordenación mediante el método sort() ya que los fragmentos pueden llegar desordenados, y antes de construir el paquete que forman hay que ordenarlos.

```
list <IPPacket*> listPackets;
```

- **Window:** Simula la ventana deslizante en un sentido de la comunicación TCP, es decir, de los paquetes TCP enviados del servidor al cliente o viceversa. Debe mantener los paquetes permanentemente en orden y por ello utiliza list.

```
list <TCPPacket*> queuePackets;
```

- **Event:** Se utilizan listas para almacenar tanto las peticiones automáticas como las manuales. Se usan listas porque facilita el acceso al conjunto de peticiones cuando hay que realizar búsquedas, eliminaciones, etc.

```
list <char*> manualURL;
list <char*> automaticURL;
```

- **NavigationSession:** Lista de eventos ordenados tal que el siguiente proviene del anterior. El último evento es el más reciente.

```
list <Event*> queueEvent;
```

13 Redes ethernet

Ethernet es la tecnología de red LAN más usada, resultando idóneas para aquellos casos en los que se necesita una red local que deba transportar tráfico esporádico y ocasionalmente pesado a velocidades muy elevadas. Las redes Ethernet se implementan con una topología física de estrella y lógica de bus, y se caracterizan por su alto rendimiento a velocidades de 10-100 Mbps.

El origen de las redes Ethernet hay que buscarlo en la Universidad de Hawai, donde se desarrollo, en los años setenta, el Método de Acceso Múltiple con Detección de Portadora y Detección de Colisiones, CSMA/CD (Carrier Sense and Multiple Access with Collition Detection), utilizado actualmente por Ethernet. Este método surgió ante la necesidad de implementar en las islas Hawai un sistema de comunicaciones basado en la transmisión de datos por radio, que se llamó Aloha, y permite que todos los dispositivos puedan acceder al mismo medio, aunque sólo puede existir un único emisor encada instante. Con ello todos los sistemas pueden actuar como receptores de forma simultánea, pero la información debe ser transmitida por turnos.

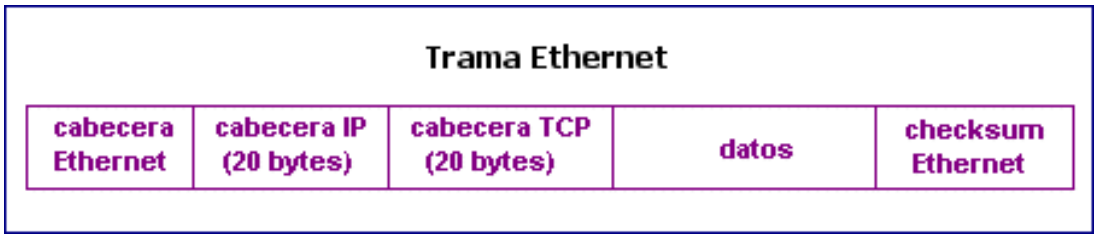
El centro de investigaciones PARC (Palo Alto Research Center) de la Xerox Corporation desarrolló el primer sistema Ethernet experimental en los años 70, que posteriormente sirvió como base de la especificación 802.3 publicada en 1980 por el Institute of Electrical and Electronic Engineers (IEEE).

Las redes Ethernet son de carácter no determinista, en la que los hosts pueden transmitir datos en cualquier momento. Antes de enviarlos, escuchan el medio de transmisión para determinar si se encuentra en uso. Si lo está, entonces esperan. En caso contrario, los host comienzan a transmitir. En caso de que dos o más host empiecen a transmitir tramas a la vez se producirán encontronazos o choques entre tramas diferentes que quieren pasar por el mismo sitio a la vez. Este fenómeno se denomina colisión, y la porción de los medios de red donde se producen colisiones se denomina dominio de colisiones. Una colisión se produce pues cuando dos máquinas escuchan para saber si hay tráfico de red, no lo detectan y, acto seguido transmiten de forma simultánea. En este caso, ambas transmisiones se dañan y las estaciones deben volver a transmitir más tarde. Para intentar solventar esta pérdida de paquetes, las máquinas poseen mecanismos de detección de las colisiones y algoritmos de postergación que determinan el momento en que aquellas que han enviado tramas que han sido destruidas por colisiones pueden volver a transmitirlas.

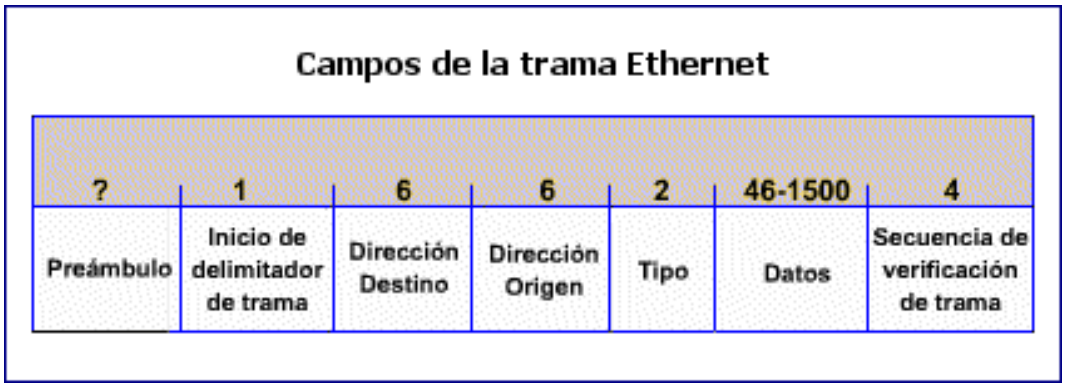
Los datos generados en la capa de aplicación pasan a la capa de transporte, que los divide en segmentos, porciones de datos aptas para su transporte por red, y luego van

descendiendo pos las sucesivas capas hasta llegar a los medios físicos. Conforme los datos van bajando por la pila de capas, paso a paso cada protocolo les va añadiendo una serie de cabeceras y datos adicionales, necesarios para poder ser enviados a su destino correctamente. El resultado final es una serie de unidades de información denominadas tramas, que son las que viajan de un host a otro.

La forma final de la trama obtenida, en redes Ethernet, es la siguiente:



Y los campos que la forman son:



- **Preámbulo**: Patrón de unos y ceros que indica a las estaciones receptoras que una trama es Ethernet o IEEE 802.3. La trama Ethernet incluye un byte adicional que es el equivalente al campo Inicio de Trama (SOF) de la trama IEEE 802.3.
- **Inicio de trama (SOF)**: Byte delimitador de IEEE 802.3 que finaliza con dos bits 1 consecutivos, y que sirve para sincronizar las porciones de recepción de trama de todas las estaciones de la red. Este campo se especifica explícitamente en Ethernet.
- **Direcciones destino y origen**: Incluye las direcciones físicas (MAC) únicas de la máquina que envía la trama y de la máquina destino. La dirección origen siempre es una dirección única, mientras que la de destino puede ser de broadcast única (trama enviada a una sola máquina), de broadcast múltiple (trama enviada a un grupo) o de broadcast (trama enviada a todos los nodos).
- **Tipo (Ethernet)**: Especifica el protocolo de capa superior que recibe los datos una vez que se ha completado el procesamiento Ethernet.

- **Longitud (IEEE 802.3)**: Indica la cantidad de bytes de datos que sigue este campo.
- **Datos**: Incluye los datos enviados en la trama. En las especificación IEEE 802.3, si los datos no son suficientes para completar una trama mínima de 64 bytes, se insertan bytes de relleno hasta completar ese tamaño (tamaño mínimo de trama). Por su parte, las especificaciones Ethernet versión 2 no especifican ningún relleno, Ethernet espera por lo menos 46 bytes de datos.
- **Secuencia de verificación de trama** (FCS): Contiene un valor de verificación CRC (Control de Redundancia Cíclica) de 4 bytes, creado por el dispositivo emisor y recalculado por el dispositivo receptor para verificar la existencia de tramas dañadas.

Cuando un paquete es recibido por el destinatario adecuado, les retira la cabecera de Ethernet y el checksum de verificación de la trama, comprueba que los datos corresponden a un mensaje IP y entonces lo pasa a dicho protocolo para que lo procese. El tamaño máximo de los paquetes en las redes Ethernet es de 1500 bytes.

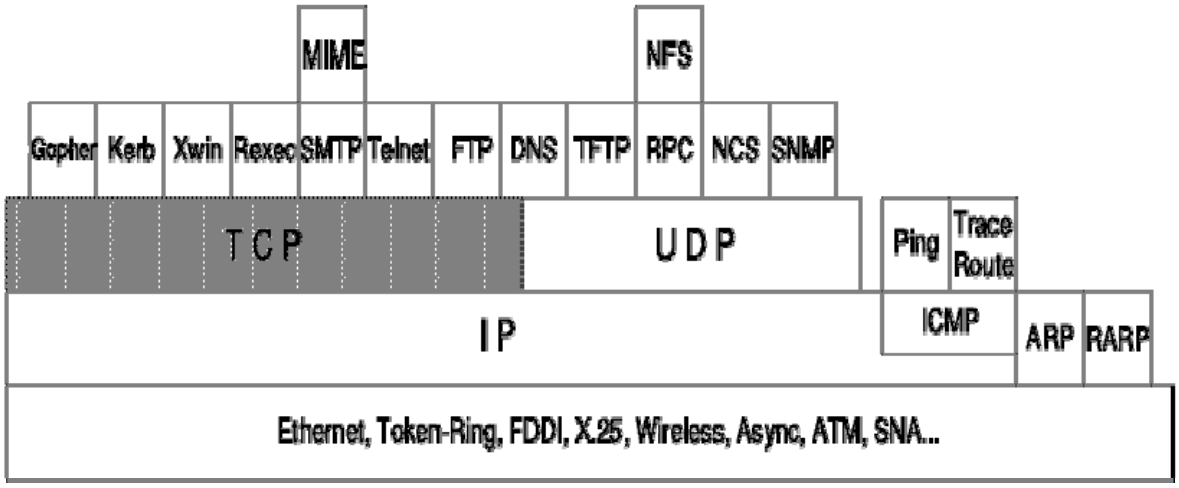
14 TCP/IP

14.1 Breve introducción histórica

El Protocolo de Internet (IP) y el Protocolo de Transmisión (TCP), fueron desarrollados inicialmente en 1973 por el informático estadounidense Vinton Cerf como parte de un proyecto dirigido por el ingeniero norteamericano Robert Kahn y patrocinado por la Agencia de Programas Avanzados de Investigación (ARPA, siglas en inglés) del Departamento Estadounidense de Defensa. Internet comenzó siendo una red informática de ARPA (llamada ARPAnet) que conectaba redes de ordenadores de varias universidades y laboratorios en investigación en Estados Unidos. World Wibe Web se desarrolló en 1989 por el informático británico Timothy Berners-Lee para el Consejo Europeo de Investigación Nuclear (CERN, siglas en francés).

14.2 ¿Qué es TCP/IP?

Es un protocolo de comunicaciones que se basa en software utilizado en redes. Aunque el nombre TCP/IP implica que el ámbito total del producto es la combinación de dos protocolos - protocolo de control de transmisión - (transmission control protocol) y protocolo Internet (Internet protocol) , el término TCP/IP no es una entidad única que combina dos protocolos, sino un conjunto de programas de software más grande que proporciona servicios de red, como registro de entrada remoto, transferencia de archivos remota y correo electrónico. TCP/IP ofrece un método de transferir información de una máquina a otra. Un protocolo de comunicaciones debe manejar los errores en la transmisión, administrar el encaminamiento y entrega de los datos, así como controlar la transmisión real mediante el uso de señales de estado predeterminadas.



Conjuntos de protocolos TCP/IP

TCP/IP es el protocolo común utilizado por todos los ordenadores conectados a Internet, de manera que éstos puedan comunicarse entre sí. Hay que tener en cuenta que en Internet se encuentran conectados ordenadores de clases muy diferentes y con hardware y software incompatibles en muchos casos, además de todos los medios y formas posibles de conexión. Aquí se encuentra una de las grandes ventajas del TCP/IP, pues este protocolo se encargará de que la comunicación entre todos sea posible. TCP/IP es compatible con cualquier sistema operativo y con cualquier tipo de hardware.

TCP/IP necesita funcionar sobre algún tipo de red o de medio físico que proporcione sus propios protocolos para el nivel de enlace de Internet. Por este motivo hay que tener en cuenta que los protocolos utilizados en este nivel pueden ser muy diversos y no forman parte del conjunto TCP/IP. Sin embargo, esto no debe ser problemático puesto que una de las funciones y ventajas principales del TCP/IP es proporcionar una abstracción del medio de forma que sea posible el intercambio de información entre medios diferentes y tecnologías que inicialmente son incompatibles.

Para transmitir información a través de TCP/IP, ésta debe ser dividida en unidades de menor tamaño. Esto proporciona grandes ventajas en el manejo de los datos que se transfieren y, por otro lado, esto es algo común en cualquier protocolo de comunicaciones. En TCP/IP cada una de estas unidades de información recibe el nombre de "datagrama",y son conjuntos de datos que se envían como mensajes independientes.

14.3 Características principales de TCP/IP

La tarea de IP es llevar los datos (paquetes) de un sitio a otro. Las computadoras que encuentran las vías para llevar los datos de una red a otra (denominadas enrutadores) utilizan IP para trasladar los datos. En resumen IP mueve los paquetes de datos, mientras TCP se encarga del flujo y asegura que los datos estén correctos.

Las líneas de comunicación se pueden compartir entre varios usuarios. Cualquier tipo de paquete puede transmitirse al mismo tiempo, y se ordenará y combinará cuando llegue a su destino.

Los datos no tienen que enviarse directamente entre dos computadoras. Cada paquete pasa de computadora en computadora hasta llegar a su destino. Lo que realmente sorprende es que sólo se necesitan algunos segundos para enviar un archivo de buen tamaño de una máquina a otra, aunque estén separadas por miles de kilómetros y pese a que los datos tienen que pasar por múltiples computadoras. Una de las razones de la

rapidez es que, cuando algo anda mal, sólo es necesario volver a transmitir un paquete, no todo el mensaje.

Los paquetes no necesitan seguir la misma trayectoria. La red puede llevar cada paquete de un lugar a otro y usar la conexión más idónea que esté disponible en ese instante. No todos los paquetes de los mensajes tienen que viajar, necesariamente, por la misma ruta, ni necesariamente tienen que llegar todos al mismo tiempo.

La flexibilidad del sistema lo hace muy fiable. Si un enlace se pierde, el sistema usa otro. Cuando se envía un mensaje, TCP divide los datos en paquetes, ordena éstos en secuencia, agrega cierta información para control de errores y después los lanza hacia fuera, y los distribuye. En el otro extremo, el TCP recibe los paquetes, verifica si hay errores y los vuelve a combinar para convertirlos en los datos originales. De haber error en algún punto, el destinatario envía un mensaje solicitando que se vuelvan a enviar determinados paquetes.

Los datagramas IP están formados por palabras de 32 bits. Cada datagrama tiene un mínimo de cinco palabras y un máximo de quince y su formato es el siguiente:

<table>
<tr><td>Ver</td><td>Hlen</td><td>TOS</td><td colspan="2">Longitud Total</td></tr>
<tr><td colspan="3">Identificación</td><td>Flags</td><td>Desplazamiento de fragmento</td></tr>
<tr><td colspan="2">TTL</td><td>Protocolo</td><td colspan="2">Checksum</td></tr>
<tr><td colspan="5">Dirección IP de la Fuente</td></tr>
<tr><td colspan="5">Dirección IP del Destino</td></tr>
<tr><td colspan="4">Opciones IP (Opcional)</td><td>Relleno</td></tr>
<tr><td colspan="5">DATOS</td></tr>
</table>

- **Ver**: Versión de IP que se emplea para construir el datagrama. Se requiere para que quien lo reciba lo interprete correctamente. La actual versión IP es la 4.
- **Hlen:** Tamaño de la cabecera en palabras de cuatro bytes.
- **TOS**: Tipo de servicio. La gran mayoría de los Host y Routers ignoran este campo. Su estructura es:

<table>
<tr><td>Prioridad</td><td>D</td><td>T</td><td>R</td><td>Sin Uso</td></tr>
</table>

La prioridad (0 = Normal, 7 = Control de red) permite implementar algoritmos de control de congestión más eficientes. Los tipos D, T y R solicitan un tipo de transporte dado: D = Procesamiento con retardos cortos, T = Alto Desempeño y R = Alta confiabilidad. Nótese que estos bits son solo "sugerencias", no es obligatorio para la red cumplirlo.

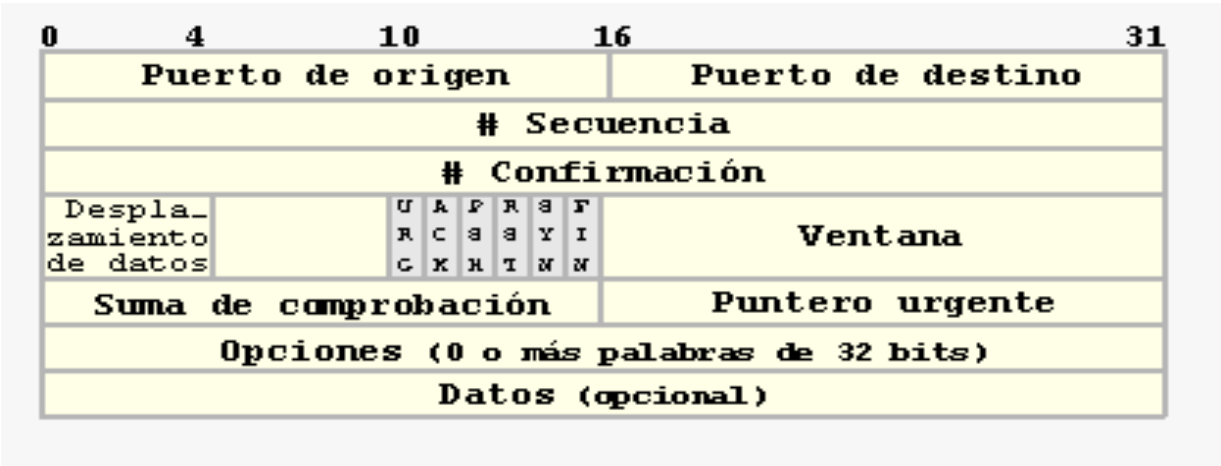
- **Longitud Total**: Mide en bytes la longitud de doto el Datagrama. Permite calcular el tamaño del campo de datos: Datos = Longitud Total – 4 * Hlen.
- **Fragmentación**. El tamaño para un datagrama debe ser tal que permita la encapsulación, esto es, enviar un datagrama completo en una trama física. El problema está en que el datagrama debe transitar por diferentes redes físicas, con diferentes tecnologías y diferentes capacidades de transferencia. A la capacidad máxima de transferencia de datos de una red física se le llama MTU (el MTU de ethernet es 1500 bytes por trama, la de FDDI es 4497 bytes por trama). Cuando un datagrama pasa de una red a otra con un MTU menor a su tamaño es necesaria la fragmentación. A las diferentes partes de un datagrama se les llama fragmento**.** Al proceso de reconstrucción del datagrama a partir de sus fragmentos se le llama Reensamblado de fragmentos. El control de la fragmentación de un datagrama IP se realiza con los campos de la segunda palabra de su cabecera:
    - **Identificación**: Numero de 16 bits que identifica al datagrama, que permite implementar números de secuencias y que permite reconocer los diferentes fragmentos de un mismo datagrama, pues todos ellos comparten este número.
    - **Banderas**: Un campo de tres bits donde el primero está reservado. El segundo, llamado bit de “No Fragmentación” es 0 si puede fragmentarse el datagrama o 1 si no puede fragmentarse el datagrama. El tercer bit es llamado “Más Fragmentos” y es 0 si es el único fragmento o último fragmento o 1 si aun hay más fragmentos. Cuando hay un 0 en “Más Fragmentos”, debe evaluarse el campo “Desplazamiento de Fragmento”, si este es cero, el datagrama no está fragmentado, si es diferente de cero, el datagrama es un último fragmento.
    - **Desp. de Fragmento**: A un trozo de datos se le llama bloque de fragmento. Este campo indica el tamaño del desplazamiento en bloques de fragmento con respecto al datagrama original, empezando por el cero.

  Para finalizar con el tema de fragmentación, hay que mencionar el Plazo de Reensamblado, que es un time out que el Host destino establece como máximo para esperar por todos los fragmentos de un datagrama. Si se vence y aun no llegan todos, entonces se descartan los que ya han llegado y se solicita el reenvío del datagrama completo.

- **TTL**: tiempo de vida del datagrama, especifica el número de segundos que se permite al datagrama circular por la red antes de ser descartado.
- **Protocolo**: Especifica que protocolo de alto nivel se empleó para construir el mensaje transportado en el campo datos de datagrama IP. Algunos valores posibles son: 1 = ICMP, 6 = TCP, 17 = UDP.
- **Checksum**: Es un campo de 16 bits que se calcula haciendo el complemento a uno de cada palabra de 16 bits del encabezado, sumándolas y haciendo su complemento a uno. Esta suma hay que recalcularla en cada nodo intermedio debido a cambios en el TTL o por fragmentación.
- **Dirección IP de la Fuente**.
- **Dirección IP del Destino**.
- **Opciones IP**: Existen hasta 40 bytes extra en la cabecera del datagrama IP que pueden llevar una o más opciones. Su uso es bastante raro. Algunas opciones son las siguientes:
    - Uso de Ruta Estricta (Camino Obligatorio)
    - Ruta de Origen Desconectada (Nodos Obligatorios)
    - Crear registro de Ruta
    - Marcas de Tiempo
    - Seguridad Básica del Departamento de Defensa
    - Seguridad Extendida del Departamento de Defensa

En los datos del datagrama IP está la información TCP. La unidad de datos de este protocolo recibe el nombre de segmento TCP. Como la cabecera debe implementar todos los mecanismos del protocolo su tamaño es bastante grande, como mínimo 20 bytes.

En la siguiente figura se puede ver el formato de la cabecera TCP:



A continuación hay una descripción de cada uno de los campos que forman la cabecera:

- **Puerto origen** (16 bits): Es el punto de acceso de la aplicación en el origen.
- **Puerto destino** (16 bits): Es el punto de acceso de la aplicación en el destino.
- **Número de secuencia** (32 bits): Identifica el primer byte del campo de datos. En este protocolo no se enumeran segmentos sino bytes, por lo que este número indica el primer byte de datos que hay en el segmento. Al principio de la conexión se asigna un número de secuencia inicial (ISN, Initial Sequence Number) y a continuación los bytes son numerados consecutivamente.
- **Número de confirmación (ACK)** (32 bits): El protocolo TCP utiliza la técnica de piggybacking para reconocer los datos. Cuando el bit ACK está activo, este campo contiene el número de secuencia del primer byte que espera recibir. Dicho de otra manera, el número ACK indica el último bit reconocido.
- **Longitud de la cabecera** (4 bits): Indica el número de palabras de 32 bits que hay en la cabecera. De esta manera el TCP puede saber donde se acaba la cabecera y por lo tanto donde empieza los datos. Normalmente el tamaño de la cabecera es de 20 bytes por lo que en este campo se almacenará el número 5. Si el TCP utiliza todos los campos de opciones la cabecera puede tener una longitud máxima de 60 bytes almacenándose en este campo el valor 15.
- **Reservado** (6 bits): Se ha reservado para su uso futuro y se inicializa con ceros.
- **Indicadores o campo de control** (6 bits): Cada uno de los bits recibe el nombre de indicador y cuando está a 1 indica una función específica del protocolo.
    - **URG:** Hay datos urgentes y en el campo "puntero urgente" se indica el número de datos urgentes que hay en el segmento.
    - **ACK:** Indica que tiene significado el número que hay almacenado en el campo "número de confirmación".
    - **PSH:** Sirve para invocar la función de carga (push). Con esta función se indica al receptor que debe pasar a la aplicación todos los datos que tenga en la memoria intermedia sin esperar a que sean completados. De esta manera se consigue que los datos no esperen en la memoria receptora hasta completar un segmento de dimensión máxima. No se debe confundir con el indicador URG que sirve para señalar que la aplicación ha determinado una parte del segmento como urgente.
    - **RST:** Sirve para hacer un reset de la conexión.
    - **SYN:** Sirve para sincronizar los números de secuencia.
    - **FIN:** Sirve para indicar que el emisor no tiene mas datos para enviar.

- **Ventana** (16 bits): Indica cuantos bytes tiene la ventana de transmisión del protocolo de control de flujo utilizando el mecanismo de ventanas deslizantes. A diferencia de lo que ocurre en los protocolos del nivel de enlace, en los que la

ventana era constante y contaba tramas, en el TCP la ventana es variable y cuenta bytes. Contiene el número de bytes de datos comenzando con el que se indica en el campo de confirmación y que el que envía está dispuesto a aceptar.

- **Suma de comprobación** (16 bits): Este campo se utiliza para detectar errores mediante el complemento a uno de la suma en módulo 2^{16} - 1 de todas las palabras de 16 bits que hay en el segmento mas una pseudo-cabecera. La pseudo-cabecera incluye los siguientes campos de la cabecera IP: dirección Internet origen, dirección Internet destino, el protocolo y un campo longitud del segmento. Con la inclusión de esta pseudo-cabecera, TCP se protege a si mismo de una transmisión errónea de IP. Esto es, si IP lleva un segmento a un computador erróneo, aunque el segmento esté libre de errores, el receptor detectará el error de transmisión.
- **Puntero urgente** (16 bits): Cuando el indicador URG está activo, este campo indica cual es el último byte de datos que es urgente. De esta manera el receptor puede saber cuantos datos urgentes llegan. Este campo es utilizado por algunas aplicaciones como telnet, rlogin y ftp.
- **Opciones** (variable): Si está presente permite añadir una única opción de entre las siguientes:

    - **Tiemstamp** para marcar en que momento se transmitió el segmento y de esta manera monitorizar los retardos que experimentan los segmentos desde el origen hasta el destino.
    - **Aumentar el tamaño de la ventana.**
    - **Indicar el tamaño máximo del segmento** que el origen puede enviar.

15 HTTP

15.1 Breve introducción histórica

Berners Lee es el responsable del nacimiento de la World Wide Web y del protocolo HTTP (Hipertext Transfer Protocol), que no de Internet. Éste comenzó el diseño de la WWW como un proyecto interno del CERN para el intercambio de información entre los científicos que había en el centro. Para ello se basó en el hipertexto y HTTP, protocolo de comunicaciones de Internet.

15.2 ¿Qué es HTTP?

Es un protocolo de nivel de aplicación, usado para intercambiar información hipermedia dentro del a WWW. Gracias a la hipermedia no solo se permitía el enlace a otros documentos html sino ahora también a sonidos, fotos, videos, etc...

A lo largo del tiempo han existido varias versiones del protocolo que son las siguientes:

- HTTP 0.9: Esta versión tiene una finalidad bien sencilla, la de transmitir datos de forma secuencial a través de la red.

- HTTP 1.0: En esta versión se soportan ya los mensajes en formato MIME, y se hace más versátil. Sin embargo conserva la generación de una conexión TCP distinta cada vez que tiene que descargar una URL, por lo que se produce en los servidores una gran saturación.

- HTTP 1.1: Esta versión es la más reciente, y es bastante más eficaz. Ahora se puede conocer las capacidades de los ordenadores que están implicados en una comunicación HTTP, además permite tener un servidor con varias ip's, permite las conexiones persistentes ahorrando recursos, etc. En esta versión una misma conexión puede albergar en si varias peticiones y respuestas, con las siguientes ventajas:
    - Se ahorra tiempo de CPU.
    - Se ahorra memoria.
    - Se ahorra tiempo ya que los paquetes son en general mas pequeños.
    - Las solicitudes y respuestas HTTP se pueden multiplexar, de manera que un cliente puede hacer muchas solicitudes al servidor y este responderle una sola vez de manera que no tenga que estar esperando una respuesta por cada solicitud.

- Se puede informar al cliente o al servidor de errores en la red sin tener que cerrar la conexión.
- No se sobrecarga el servidor al disminuir en gran cantidad los paquetes TCP que por el viajan.

15.3 Características principales de HTTP 1.1

Un mensaje HTTP consiste en una petición de un cliente al servidor y en la respuesta del servidor al cliente. Las peticiones y respuestas pueden ser simples o completas. La diferencia es que en las peticiones y respuestas completas se envían cabeceras y un contenido. Este contenido se pone después de las cabeceras dejando una línea vacía entre las cabeceras y el contenido. En el caso de peticiones simples, sólo se puede usar el método GET y no hay contenido. Si se trata de una respuesta simple, entonces ésta sólo consta de contenido. Esta diferenciación entre simples y completas se tiene para que el protocolo HTTP/1.0 pueda atender peticiones y enviar respuestas del protocolo HTTP/0.9.

15.4 Petición HTTP

Una petición de un cliente a un servidor ha de incluir el método que se aplica al recurso, el identificador del recurso y la versión del protocolo que usa para realizar la petición.

Una petición se haga con el protocolo HTTP/1.0 o con el protocolo HTTP/1.1 la petición sigue el formato:

Línea de petición

*(Cabeceras)

CRLF

Contenido

La línea de petición comienza indicando el método, seguido de un espacio en blanco (SP) seguido de la URI de la petición y la versión del protocolo, finalizando la línea con CRLF:

Método SP URI de la petición SP Versón de protocolo CRLF

En el caso del protocolo HTTP/0.9 sólo se permite el método GET, con el protocolo HTTP/1.0 GET, POST y HEAD y con el protocolo HTTP/1.1 OPTIONS, GET, HEAD, POST, PUT, DELETE y TRACE. En caso de que un servidor tenga implementado un

método, pero no está permitido para el recurso que se pide, entonces ha de devolver un código de estado 405 (método no permitido). Si lo que ocurre es que no tiene implementado el método, entonces devuelve un código 501 (no implementado). Los únicos métodos que deben soportar los servidores de forma obligatoria son los métodos GET y HEAD.

Existe tres tipos de cabeceras: cabeceras generales, de petición y de entidad. Las cabeceras generales son las que se aplican tanto a peticiones como a respuestas, pero no al contenido que se transmite. Las cabeceras de petición permiten al cliente pasar información al servidor sobre la petición y sobre el cliente. Las cabeceras de entidad permiten definir información adicional sobre el contenido que se transmite y en caso de que no haya contenido, sobre el recurso al que se quiere acceder con la petición.

El contenido (si está presente) está en un formato con una codificación definida en las cabeceras de entidad.

15.5 Respuesta HTTP

Después de recibir e interpretar una petición, el servidor debe responder con un mensaje HTTP. Este mensaje tiene el siguiente formato:

Línea de estado

*(Cabeceras)

CRLF

Contenido

La línea de estado es la primera línea de la respuesta y consiste en la versión de protocolo que se utiliza, seguida de una indicación de estado numérica a la que puede ir asociada una frase explicativa. El formato es el siguiente:

Versión del protocolo SP Código de estado SP Frase explicativa CRLF

El código de estado es un número de 3 dígitos que indica si la petición ha sido atendida satisfactoriamente o no, y en caso de no haber sido atendida, indica la causa. Los códigos se dividen en cinco clases definidas por el primer dígito del código de estado. Así tenemos:

- 1xx: Informativo. Esta clase de código de status indica una respuesta provisional, consistente únicamente en una linea de estado y cabecera opcional,

y termina con una linea vacía. No hay cabeceras obligatorias para esta clase de código de status. Como http/1.0 no define 1xx, los servidores no deben enviar una respuesta 1xx a un cliente http/1.0 salvo en casos experimentales. Un cliente debe estar preparado para aceptar una o más respuestas 1xx, incluso si no las espera. Estos códigos pueden ser ignorados.

- 2xx: Éxito. La acción requerida por la petición ha sido recibida, entendida y aceptada.
- 3xx: Redirección. Esta clase de código de status indica que acción tiene que realizar el agente para cumplir con la petición. La acción requerida puede ser llevada a cabo por el agente sin interacción del usuario, si y solo si, el método usado en la segunda petición es GET o HEAD. Un cliente debería detecta los bucles de redirecciones infinitos.
- 4xx: Error del cliente. Esta clase se utiliza en casos en los que el cliente parece haberse equivocado. Excepto cuando responde a una petición HEAD, el servidor debería incluir en la respuesta una explicación de la situación de error, y si es un fallo temporal o permanente. Si el cliente está enviando datos, el servidor que usa TCP debería tener cuidado y asegurar que el cliente recibe el paquete que contiene la respuesta, antes de que el servidor cierre la conexión. Si el cliente continua enviando datos después del cierre, la pila TCP del servidor enviará una paquete reset al cliente.
- 5xx: Error del servidor. Esta clase se utiliza en los casos en los que el servidor es consciente de que ha cometido un error o es incapaz de atender la petición. Excepto cuando responde a una petición HEAD, el servidor debería incluir en el paquete una explicación de la situación de error ,y si es un fallo temporal o permanente.

Algunos de los códigos más comúnmente usados y las frases asociadas son:

- 100, continuar.
- 101, cambio de protocolo.
- 200, éxito.
- 201, creado.
- 202, aceptado.
- 204, sin contenido.
- 300, múltiples elecciones.
- 301, movido permanentemente.
- 302, movido temporalmente.
- 400, petición errónea.
- 401, no autorizado.
- 402, pago requerido.
- 403, prohibido.

- 404, no encontrado.
- 405, método no permitido.
- 406, no se puede aceptar.
- 408, límite de tiempo de la petición.
- 413, contenido de la petición muy largo.
- 414, URI de la petición muy largo.
- 500, error interno del servidor.
- 501, no implementado.
- 502, puerta de enlace errónea.
- 503, servicio no disponible.
- 504, tiempo límite de la puerta de enlace.
- 505, versión de protocolo HTTP no soportada.

La frase explicativa, es una frase corta que explica el código de estado enviado al cliente. Se pueden usar más códigos, pero las aplicaciones HTTP no tienen que conocer todos los códigos definidos y su significado, pero sí están obligadas a conocer su clase y tratar los códigos desconocidos como el primer código de la clase (x00).

Existen tres tipos de cabeceras de la respuesta, son de tres tipos: las cabeceras generales, las cabeceras de la respuesta y las cabeceras de entidad.

Las cabeceras de respuesta permiten al servidor enviar información adicional al cliente sobre la respuesta. Estos campos dan información sobre el servidor y acceso al recurso pedido.

15.6 Métodos HTTP

Un método se dice que es seguro si no provocan ninguna otra acción que no sea la de devolver algo (no produce efectos laterales). Estos métodos son el método GET y el método HEAD. Para realizar acciones inseguras (las que afectan a otras acciones) se pueden usar los métodos POST, PUT y DELETE. Aunque esto está definido así, no se puede asegurar que un método seguro no produzca efectos laterales, porque depende de la implementación del servidor.

Un método es idempotente si los efectos laterales para N peticiones son los mismos que para una sola petición. Los métodos idempotentes son los métodos GET, HEAD, PUT y DELETE.

Método OPTIONS: este método representa un petición de información sobre las opciones de comunicación disponibles en la cadena petición-respuesta identificada por

la URI de la petición. Esto permite al cliente conocer las opciones y requisitos asociados con un recurso o las capacidades del servidor. La respuesta sólo debe incluir información sobre las opciones de comunicación. Si la URI es "*"', entonces la petición se aplica al servidor como un conjunto. Es decir, contesta características opcionales definidas por el servidor, extensiones del protocolo, etc...

Método GET: este método requiere la devolución de información al cliente identificada por la URI. Si la URI se refiere a un proceso que produce información, se devuelve la información y no la fuente del proceso. El método GET pasa a ser un GET condicional si la petición incluye las cabeceras If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match o If-Range. Estas cabeceras hacen que el contenido de la respuesta se transmita sólo si se cumplen unas condiciones determinadas por esas cabeceras. Esto se hizo para reducir el tráfico en las redes. También hay un método GET parcial, con el que se envía sólo parte del contenido del recurso requerido. Esto ocurre cuando la petición tiene una cabecera Range. Al igual que el método GET condicional, el método GET parcial se creó para reducir el tráfico en la red.

Método HEAD: este es igual que el método GET, salvo que el servidor no tiene que devolver el contenido, sólo las cabeceras. Estas cabeceras que se devuelven en el método HEAD deberían ser las mismas que las que se devolverían si fuese una petición GET. Este método se puede usar para obtener información sobre el contenido que se va a devolver en respuesta a la petición. Se suele usar también para chequear la validez de links, accesibilidad y modificaciones recientes.

Método POST: este método se usa para hacer peticiones en las que el servidor destino acepta el contenido de la petición como un nuevo subordinado del recurso pedido. El método POST se creó para cubrir funciones como la de enviar un mensaje a grupos de usuarios, dar un bloque de datos como resultado de un formulario a un proceso de datos, añadir nuevos datos a una base de datos, etc... La función llevada a cabo por el método POST está determinada por el servidor y suele depender de la URI de la petición. El resultado de la acción realizada por el método POST puede ser un recurso que no sea identificable mediante una URI.

Método PUT: este método permite guardar el contenido de la petición en el servidor bajo la URI de la petición. Si esta URI ya existe, entonces el servidor considera que esta petición proporciona una versión actualizada del recurso. Si la URI indicada no existe y es válida para definir un nuevo recurso, el servidor puede crear el recurso con esa URI. Si se crea un nuevo recurso, debe responder con un código 201 (creado), si se modifica se contesta con un código 200 (OK) o 204 (sin contenido). En caso de que no se pueda crear el recurso se devuelve un mensaje con el código de error apropiado. La principal

diferencia entre POST y PUT se encuentra en el significado de la URI. En el caso del método POST, la URI identifica el recurso que va a manejar en contenido, mientras que en el PUT identifica el contenido. Un recurso puede tener distintas URI.

Método DELETE: este método se usa para que el servidor borre el recurso indicado por la URI de la petición. No se garantiza al cliente que la operación se lleve a cabo aunque la respuesta sea satisfactoria.

Método TRACE: este método se usa para saber si existe el receptor del mensaje y usar la información para hacer un diagnóstico. En las cabeceras el campo Via sirve para obtener la ruta que sigue el mensaje. Mediante el campo Max-Forwards se limita el número de pasos intermedios que puede tomar. Esto es útil para evitar bucles entre los proxy. La petición con el método TRACE no tiene contenido.

15.7 Cabecera HTTP

Como comentamos anteriormente existen cuatro tipos de cabecera: general, petición, respuesta y entidad.

- Cabeceras generales: los campos de este tipo de cabeceras se aplican tanto a las peticiones como a las respuestas, pero no al contenido de los mensajes. Los diferentes cabeceras generales son:
    - Cache-Control: son directivas que se han de tener en cuenta a la hora de mantener el contenido en una caché.
    - Connection: permite especificar opciones requeridas para una conexión.
    - Date: representa la fecha y la hora a la que se creó el mensaje.
    - Pragma: usado para incluir directivas de implementación.
    - Transfer-Encoding: indica la codificación aplicada al contenido.
    - Upgrade: permite al cliente especificar protocolos que soporta.
    - Via: usado por pasarelas y *proxies* para indicar los pasos seguidos.
- Cabeceras de petición: este tipo de cabeceras permite al cliente pasar información adicional al servidor sobre la petición y el propio cliente. Las diferentes cabeceras de petición son:
    - Accept: indican el tipo de respuesta que acepta.
    - Accept-Charset: indica los conjuntos de caracteres que acepta.
    - Accept-Encoding: que tipo de codificación acepta.
    - Accept-Language: tipo de lenguaje de la respuesta que se prefiere.
    - Authorization: el agente de usuario quiere autentificarse con el servidor.

  - From: contiene la dirección de correo que controla en agente de usuario.
  - Host: especifica la máquina y el puerto del recurso pedido.
  - If-Modified-Since: para el GET condicional.
  - If-Match: para el GET condicional.
  - If-None-Match: para el GET condicional.
  - If-Range: para el GET condicional.
  - If-Unmodified-Since: para el GET condicional.
  - Max-Forwards: indica el máximo número de elementos por los que pasa.
  - Proxy-Authorization: permite que el cliente se identifique a un *proxy*.
  - Range: establece un rango de *bytes* del contenido.
  - Referer: indica la dirección donde obtuvo la URI de la petición.
  - User-Agent: información sobre el agente que genera la petición.
- Cabeceras de respuesta: permiten al servidor pasar información adicional al cliente sobre la respuesta, el propio servidor y el recurso solicitado. Las diferentes cabeceras de respuesta son:
  - Age: estimación del tiempo transcurrido desde que se creó la respuesta.
  - Location: se usa par a redirigir la petición a otra URI.
  - Proxy-Authenticate: ante una respuesta con el código 407 (autentificación *proxy* requerida), indica el esquema de autentificación.
  - Public: da la lista de métodos soportados por el servidor.
  - Retry-After: ante un servicio no disponible da una fecha para volver a intentarlo.
  - Server: información sobre el servidor que maneja las peticiones.
  - Vary: indica que hay varias respuestas y el servidor ha escogido una.
  - Warning: usada para aportar información adicional sobre el estado de la respuesta.
  - WWW-Authenticate: indica el esquema de autentificación y los parámetros aplicables a la URI.
- Cabeceras de entidad: como su nombre indica, los campos de este tipo aportan información sobre el contenido del mensaje o si no hay contenido, sobre el recurso al que hace referencia la URI de la petición. Las diferentes cabeceras de entidad son:
  - Allow: da los métodos soportados por el recurso designado por la URI.
  - Content-Base: indica la URI base para resolver las URI relativas.

- Content-Encoding: indica una codificación adicional aplicada al contenido (a parte de la aplicada por el tipo).
- Content-Language: describe el idioma del contenido.
- Content-Length: indica el tamaño del contenido del mensaje.
- Content-Location: da información sobre la localización del recurso que da el contenido del mensaje.
- Content-MD5: es un resumen en formato MD5 (RFC 1864) para chequear la integridad del contenido.
- Content-Range: en un GET parcial, indica la posición del contenido.
- Content-Type: indica el tipo de contenido que es.
- Etag: define una marca para el contenido asociado.
- Expires: indica la fecha a partir de la cual la respuesta deja de ser válida.
- Last-Modified: indica la fecha de la última modificación.

16.Bibliografía

- Dr. Sidnie Feit (1998) TCP/IP. Arquitectura, protocolos, implementación y seguridad con Ipv6 y seguridad Ip. Ed. McGraw-Hill. ISBN 84-481-1351.

- Andrew S. Tanenbaum (1997) Redes de computadores. Ed. Prentice Hall. ISBN 968-880-958-6.

- Douglas E. Comer (1996) Redes globales de información con Internet y TCP/IP. Principios básicos, protocolos y arquitectura. Ed. Prentice Hall. ISBN 968-880-541-6.

- Charles E. Spurgeon (2000) Ethernet: The definitive guide. Ed. O´Reilly. ISBN 1-56592-6609.

- F.M. Márquez. 1996. UNIX: Programación avanzada. 2ª Ed. Ra-Ma

Documentación utilizada a partir de la web:

- API de la librería STL.
  http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vcstdlib/html/vcoriHeaders_STL.asp

- Información sobre el uso de hilos en C, C++.
  http://webdia.cem.itesm.mx/ac/rogomez/Apuntes/artiThreads.ps

- Información sobre protocolos TCP, IP.
  http://club.telepolis.com/jlrosalesf/FUNDAMENTOS%20DEL%20TCP%20-2-.htm

- Tipos de datos en C, C++.
  http://www.programacionfacil.com/linuxc/cuatro1.htm

- Pagina oficial de pcap.
  http://tcpdump.org

- RFC en español.
  http://www.rfc-es.org/

- RFC en inglés.
  http://sscript.infdj.com/IRC/Protocolos/http-rfc2616-.txt

17. Palabras clave de referencia

- Sniffer.
- Monitorear.
- Pcap.
- Ethereal.
- TCP/IP.
- HTTP.
- Ethernet.
- Espiar.