3 Práctica 2: Análisis del protocolo TPC/IP - Wireshark

3.1 Objetivo

En esta práctica los alumnos analizarán mediante el software Wireshark (analizador de red) distintas tramas envíadas o recibidas en el PC. El objetivo es ver y analizar tramas reales de tráfico y ver su correspondencia con los niveles OSI asociados al protocolo TCP/IP. El alumno debe entregar una pequeña memoria en word con el ánalisis de una trama.

3.2 Teoría de la Práctica

3.2.1 ¿Qué es un protocolo?

Un protocolo de comunicaciones es un conjunto de reglas que deben ser cumplidas por todos los usuarios en una red para poder llevar a cabo la transmisioón de datos entre dos o más puntos de la red (llamados nodos). Asociado a las citadas "*reglas*" aparece el concepto de "*capas*". Las capas o niveles de los protocolos de comunicaciones fueron creados con el objetivo de poder dividir la compleja funcionalidad de un protocolo en varios trozos más manejables. El concepto es similar a la jerarquía de organización que puede haber en una empresa. Cada capa es independiente de las otras capas y se relaciona con otras capas por medio de unas *interfaces* claramente definidas.

Una de las consecuencias principales de esta división en capas de un protocolo de comunicaciones es que los desarrolladores software se pueden concentrar en escribir código para una capa en concreto abstrayendose del resto. Así todo lo que recibe un desarrollados a la hora de implementar una capa de un protocolo es un conjunto de interfaces (funciones o métodos de clases en términos de software) que por un lado deben implementar para dar servicio a protocolos de nivel superior y por otro debe hacer uso para emplear protocolos de nivel inferior.

3.2.2 Introducción a las capas del protocolo TCP/IP

El **protocolo IP (capa de red)** permite enlazar diferentes nodos de distintas redes locales entre sí. Para ello es necesario que cada nodo tenga un identificador único (dirección IP). El protocolo IP puede hacer uso de distintas capas de **nivel de enlace** (por ejemplo, wireless, **Ethernet,** Infrarojos,..). Lo más común es que haga uso de la capa de nivel de enlace Ethernet. A su vez el protocolo Ip da servicio a otros protocolos de **nivel de transporte como TPC o UDP** y estos a su vez a otros de **aplicación como HTTP.**

A continuación, se muestra el contenido de las tramas de información en las diferentes capas o niveles OSI:

Conectando un osciloscopio al cableado de RED se ven los datos a nivel físico. Como se puede ver en ese nivel los bits son señales de tensión. **Este sería el nivel FÍSICO.**

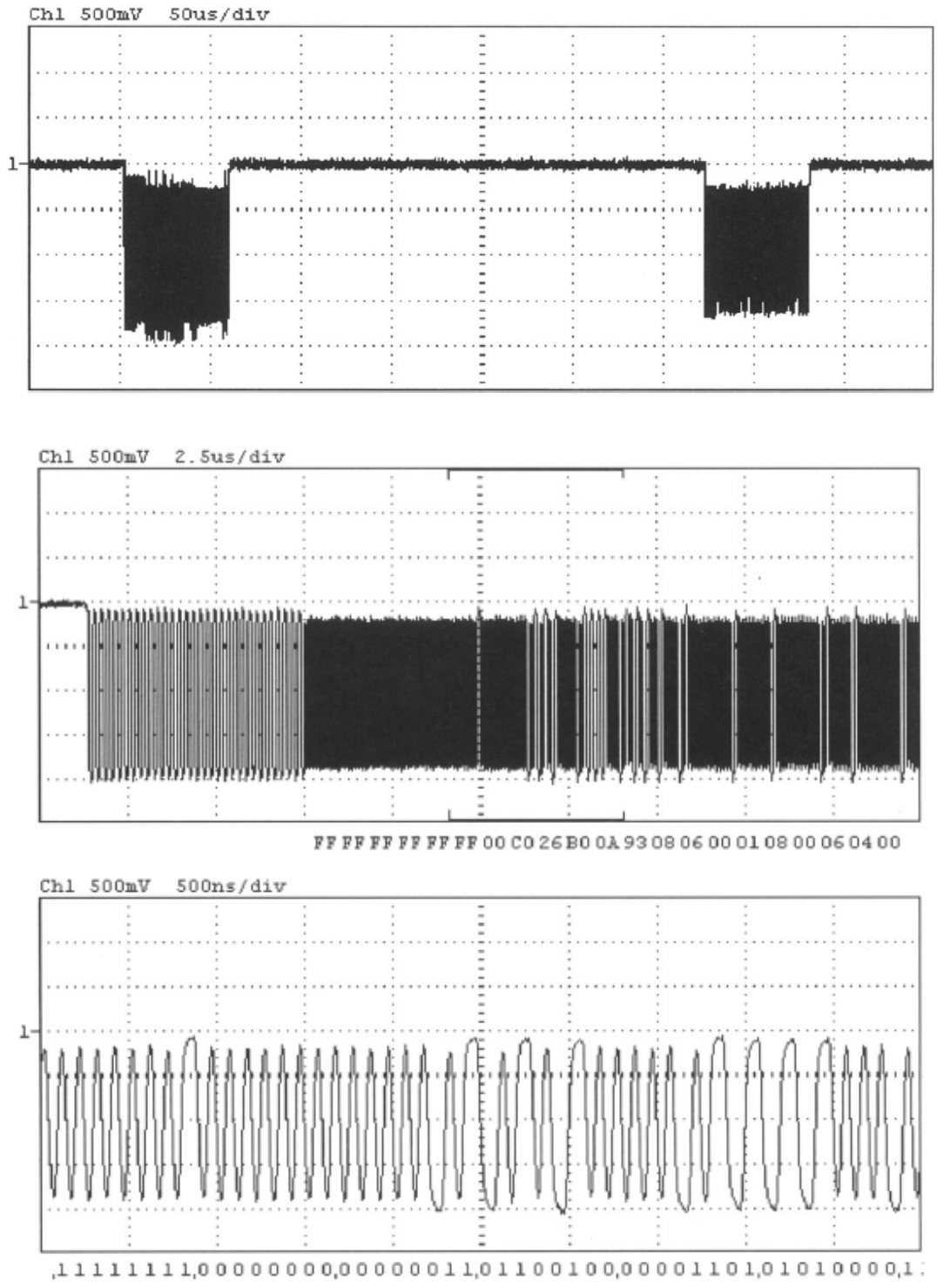

Figura 2.7 Tramas capturadas mediante un Osciloscopio

Por encima del nivel físico estaría el **nivel de enlace**. En las figuras Figura 2.8 se puede ver la descripción de los bits transmitidos a ese nivel. Es decir, los bits transmitidos en el nivel físico se agrupan para formar lo que es el protocolo de nivel de enlace.

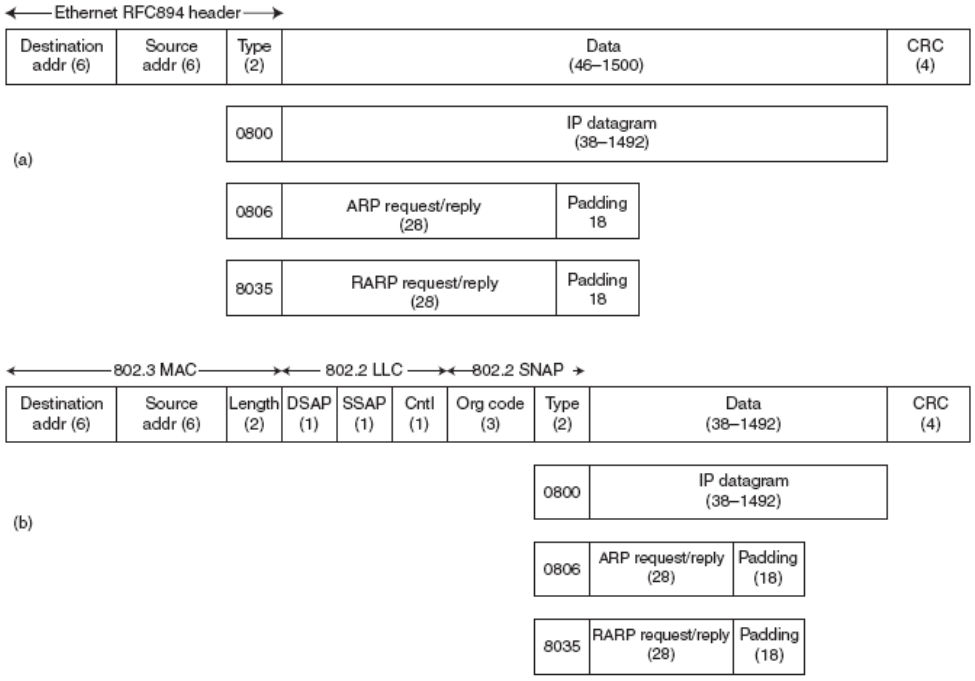

Figura 2.8 Tramas EthernetII y tramas Ethernet 802

En las figuras Figura 2.9 y Figura 2.10 se puede ver cómo la trama de datos IP se empaqueta en la zona de datos de la trama 802.2.

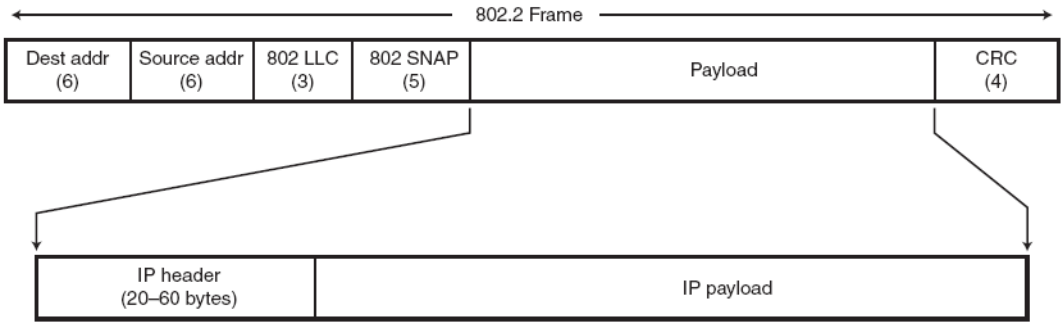

Figura 2.9 Datagrama IP y su relación con una trama IEEE 802

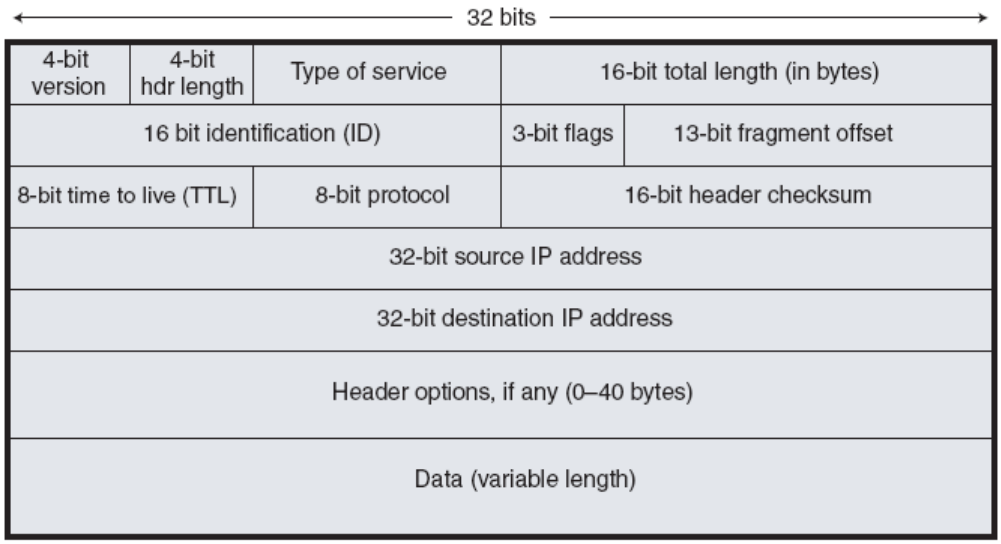

Figura 2.10 IP header (cabecera)

Por último, en la figuras Figura 2.11 se puede ver cómo se empaqueta la trama TCP en el área de datos de la trama IP.

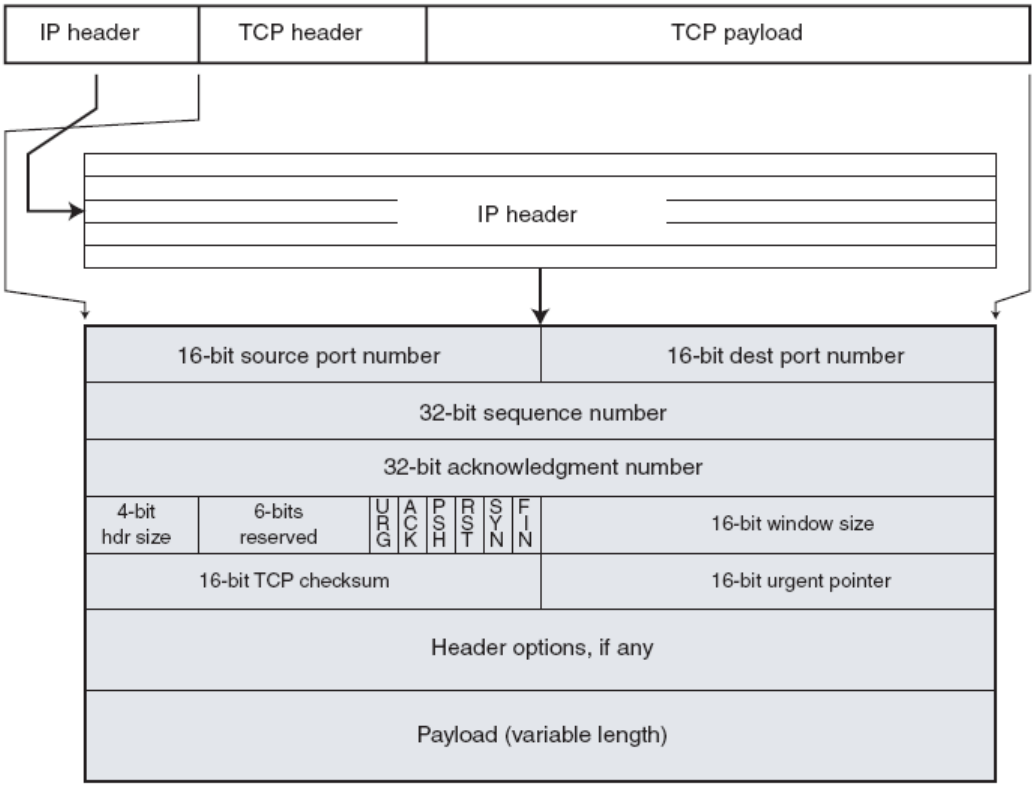

Figura 2.11 TCP header (cabecera)

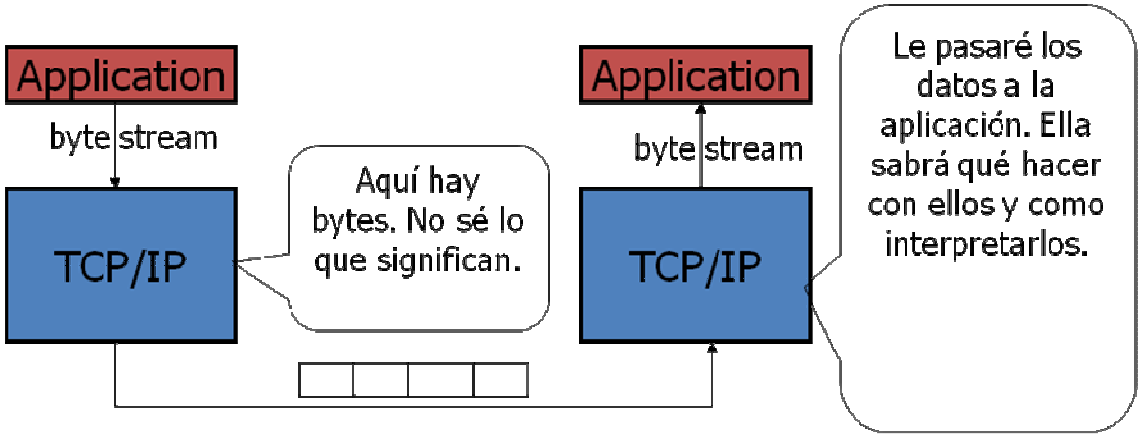

Figura 2.12 TCP/IP transporta datos

Los datos **HTTP (protocolo de aplicación)** van en la parte de datos de la trama TCP. Así, se puede entender que el procolo HTTP hace uso de los servicios proporcionados por TCP que hace uso a su vez del IP que hace uso a su vez de Ethernet.

3.2.3 ¿Cómo se traduce el concepto de capas en la realidad?

Cuando se estudian protocolos y el concepto de niveles o capas conviene tener claro en qué se traducen en la realidad conceptos como "interface" de un protocolo. Así, la interface de un protocolo por ejemplo como TCP/IP para un desarrollador de nivel de aplicación no es más que una librería de funciones o clases (API) que otros desarrolladores han creado y ponen a disposición suyo e implementa el protocolo TCP/IP. El desarrollador de nivel de aplicación no necesita saber ni tener el código fuente de dichas funciones de interface tan sólo tener claro cada función de interface qué parámetros necesita y que resultados devuelve. Por ello, las APIs de software se entregan con un manual o descripción de cara a facilitar el uso a otros desarrolladores. Librerias como las que se describen en MSDN (.NET) o otras de JAVA entregan junto con la propia interfaz una ayuda y ejemplos de uso. En la práctica es necesario por parte de los desarrolladores dedicar tiempo para entender y poder hacer hacer uso de dichas interfaces.

A continuación se describe la interface de la clase socket de Java asociado a la capa TCP.

java.net

Class Socket

java.lang.Object
  └ **java.net.Socket**

Direct Known Subclasses:
SSLSocket

---

public class **Socket**
extends Object

This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.

The actual work of the socket is performed by an instance of the `SocketImpl` class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.

Since:

JDK1.0

See Also:

`setSocketImplFactory(java.net.SocketImplFactory)`, `SocketImpl`, `SocketChannel`

Constructor Summary

	Socket() Creates an unconnected socket, with the system-default type of SocketImpl.
	Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
	Socket(InetAddress host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(InetAddress address, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote address on the specified remote port.
protected	**Socket**(SocketImpl impl) Creates an unconnected Socket with a user-specified SocketImpl.
	Socket(String host, int port) Creates a stream socket and connects it to the specified port number on the named host.
	Socket(String host, int port, boolean stream) **Deprecated.** *Use DatagramSocket instead for UDP transport.*
	Socket(String host, int port, InetAddress localAddr, int localPort) Creates a socket and connects it to the specified remote host on the specified remote port.

Method Summary

void	**bind**(SocketAddress bindpoint) Binds the socket to a local address.
void	**close**() Closes this socket.

void	**connect**(SocketAddress endpoint) Connects this socket to the server.
void	**connect**(SocketAddress endpoint, int timeout) Connects this socket to the server with a specified timeout value.
SocketChannel	**getChannel**() Returns the unique SocketChannel object associated with this socket, if any.
InetAddress	**getInetAddress**() Returns the address to which the socket is connected.
InputStream	**getInputStream**() Returns an input stream for this socket.
boolean	**getKeepAlive**() Tests if SO_KEEPALIVE is enabled.
InetAddress	**getLocalAddress**() Gets the local address to which the socket is bound.
int	**getLocalPort**() Returns the local port to which this socket is bound.
SocketAddress	**getLocalSocketAddress**() Returns the address of the endpoint this socket is bound to, or null if it is not bound yet.
boolean	**getOOBInline**() Tests if OOBINLINE is enabled.
OutputStream	**getOutputStream**() Returns an output stream for this socket.
int	**getPort**() Returns the remote port to which this socket is connected.
int	**getReceiveBufferSize**() Gets the value of the SO_RCVBUF option for this Socket, that is the buffer size used by the platform for input on this Socket.
SocketAddress	**getRemoteSocketAddress**() Returns the address of the endpoint this socket is connected to, or null if it is unconnected.
boolean	**getReuseAddress**() Tests if SO_REUSEADDR is enabled.
int	**getSendBufferSize**() Get value of the SO_SNDBUF option for this Socket, that is the buffer size used by the platform for output on this Socket.

int	**getSoLinger**() Returns setting for SO_LINGER.
int	**getSoTimeout**() Returns setting for SO_TIMEOUT.
boolean	**getTcpNoDelay**() Tests if TCP_NODELAY is enabled.
int	**getTrafficClass**() Gets traffic class or type-of-service in the IP header for packets sent from this Socket
boolean	**isBound**() Returns the binding state of the socket.
boolean	**isClosed**() Returns the closed state of the socket.
boolean	**isConnected**() Returns the connection state of the socket.
boolean	**isInputShutdown**() Returns wether the read-half of the socket connection is closed.
boolean	**isOutputShutdown**() Returns wether the write-half of the socket connection is closed.
void	**sendUrgentData**(int data) Send one byte of urgent data on the socket.
void	**setKeepAlive**(boolean on) Enable/disable SO_KEEPALIVE.
void	**setOOBInline**(boolean on) Enable/disable OOBINLINE (receipt of TCP urgent data) By default, this option is disabled and TCP urgent data received on a socket is silently discarded.
void	**setReceiveBufferSize**(int size) Sets the SO_RCVBUF option to the specified value for this Socket.
void	**setReuseAddress**(boolean on) Enable/disable the SO_REUSEADDR socket option.
void	**setSendBufferSize**(int size) Sets the SO_SNDBUF option to the specified value for this Socket.
static void	**setSocketImplFactory**(SocketImplFactory fac) Sets the client socket implementation factory for the application.

void	**setSoLinger**(boolean on, int linger) Enable/disable SO_LINGER with the specified linger time in seconds.
void	**setSoTimeout**(int timeout) Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.
void	**setTcpNoDelay**(boolean on) Enable/disable TCP_NODELAY (disable/enable Nagle's algorithm).
void	**setTrafficClass**(int tc) Sets traffic class or type-of-service octet in the IP header for packets sent from this Socket.
void	**shutdownInput**() Places the input stream for this socket at "end of stream".
void	**shutdownOutput**() Disables the output stream for this socket.
String	**toString**() Converts this socket to a String.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Constructor Detail

Socket

```
public Socket()
```

Creates an unconnected socket, with the system-default type of SocketImpl.

Since:

JDK1.1

Socket

```
protected Socket(SocketImpl impl)
          throws SocketException
```

Creates an unconnected Socket with a user-specified SocketImpl.

Parameters:

`impl` - an instance of a **SocketImpl** the subclass wishes to use on the Socket.

Throws:

SocketException - if there is an error in the underlying protocol, such as a TCP error.

Since:

JDK1.1

3.3 Análisis de tráfico mediante Wireshark

El alumno debe acceder a la dirección www.ehu.es/correow e introducir un nombre de usuario ficticio. Previamente mediante el software Wireshark debe comenzar a capturar el tráfico de red de su ordenador. Una vez introducido el nombre y pulsado enter debe parar la captura de tramas. Se debe buscar y analizar la trama donde se ha enviado el usuario.

Para iniciar la captura de tramas con Wireshark: *Capture -> Start* (seleccionado la tarjeta de red que lleva el tráfico). Para detener la captura de tramas: *Capture -> Stop*. El analizador software wireshark permite filtras por tipo de tráfico (HTTP), direcciónes IP, etc.

En la Figura 2.13 se puede ver que el usuario "hola hola" que se relleno al hacer el ejemplo en el campo usuario se transmite sin ningún tipo de cifrado. Si la página web no hiciera uso de cifrado de password este podría leerse con total facilidad. De ahí, la importancia por ejemplo de emplear https para el acceso a webs y de evitar enviar datos como tarjetas de crédito, etc. en emails que van sin cifrar.

TAREA DE LA PRÁCTICA

El alumno debe entregar una pequeña memoria donde se debe llevar a cabo el análisis y explicación de todos los campos e información de cabeceras de una trama capturada explicando qué es lo que se transmite en cada campo.

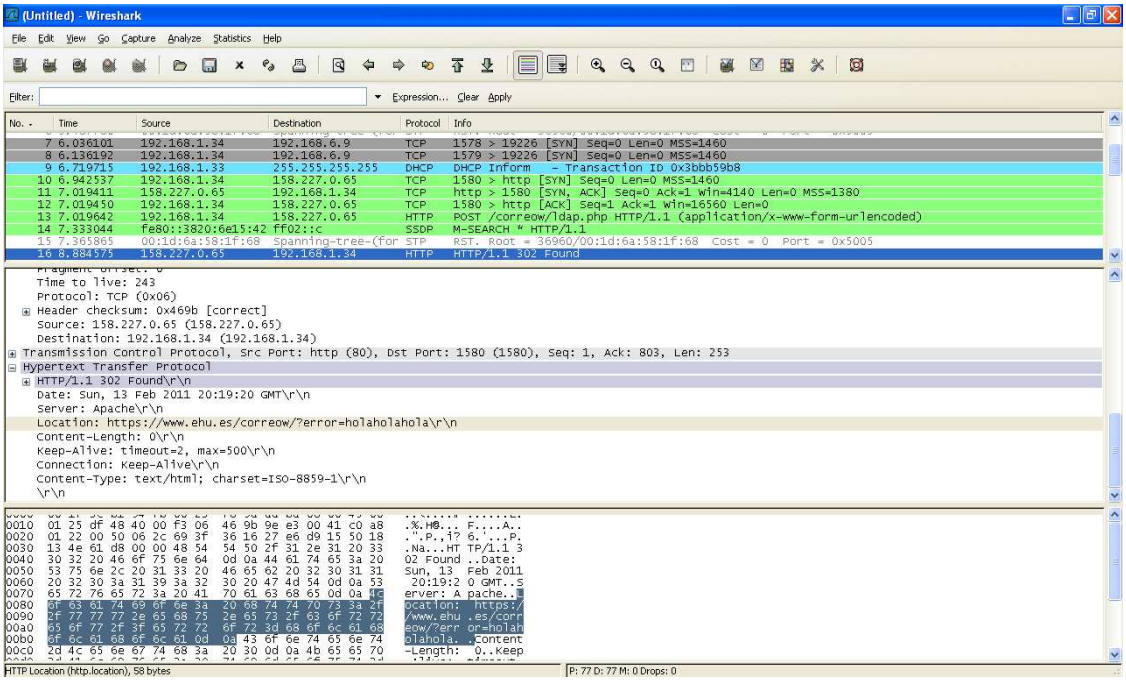

Figura 2.13 Captura de tramas con wireshark