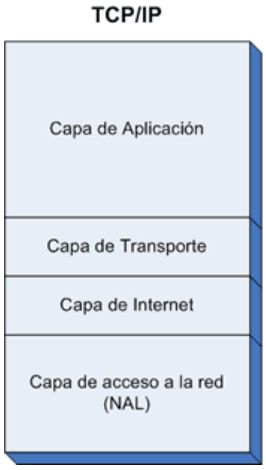
0. Introducción a la programación orientada a Internet.

1. TCP/IP. Modelo Cliente-Servidor.



La familia de protocolos ***TCP/IP*** se estructura en cuatro niveles o capas, de tal forma que los niveles más bajos se encargan de los asuntos relacionados con el hardware de red y los más altos están más cercanos a las aplicaciones de usuario. La familia toma el nombre de los dos protocolos más importantes de los niveles intermedios: *TCP* e *IP*. Dichos protocolos son los que se usan de forma estándar en Internet, y por extensión, en la gran mayoría de las redes actuales.

La capa de acceso a la red está relacionada con la conexión física a la red – interfaz de red, conexiones, tipos de señal, etc – y con la transferencia de secuencias de bits entre equipos conectados a un mismo medio. El protocolo de ésta capa depende, por tanto, de la infraestructura de red a la que está conectado el equipo, y será independiente de los protocolos de los niveles superiores. El protocolo más habitual en éste nivel es Ethernet en sus distintas modalidades – par trenzado, wireless, fibra óptica -.

En la capa de ***InterRed*** – InterNet, en inglés – se encuentra el protocolo *IP* – Internet Protocol -, que se encarga de conectar equipos de distintas redes, siempre que éstas estén interconectadas. Para ello *IP* soluciona el direccionamiento de red y el encaminamiento o *routing*. Una dirección IP identifica de forma única a un equipo o *host* en Internet. Cada paquete de datos que viaja por la red lleva una *cabecera IP* que, entre otras cosas, contiene la *dirección IP* de destino y la de origen. De esta forma los *routers* pueden utilizar esa información para encaminar cada paquete hacia su destino.

El protocolo más importante del ***nivel de transporte*** es *TCP* – Transmission Control Protocol – que se encarga de varias funciones como garantizar una comunicación fiable, realizar direccionamiento de nivel de transporte y la segmentación / reensamblaje de datos. Hay que tener en cuenta que el protocolo de nivel inferior – *IP* – no asegura una comunicación fiable, ya que algunos paquetes pueden perderse, repetirse o llegar desordenados, ya que cada paquete IP viaja por la red de forma independiente. Para garantizar una comunicación fiable, el protocolo TCP añade en las cabeceras de sus paquetes un *número de secuencia* para cada paquete de una conexión, y mediante un mecanismo de respuestas de confirmación y reenvios soluciona el problema de las pérdidas de paquetes. Si llegan paquetes desordenados, éstos se detectan en el *host* de destino y se reordenan. En el caso de que un paquete llegue repetido, simplemente se ignora y se envía una respuesta de confirmación.

El direccionamiento a nivel de transporte se lleva a cabo mediante los *puertos TCP*, que identifican a las aplicaciones de red dentro de cada *host*. Así, aunque tengamos más de una aplicación realizando conexiones de red en un equipo, cada una de ellas tendrá asignado un número de puerto distinto, de tal forma que el protocolo *TCP* podrá identificar a qué aplicación pertenece cada paquete de datos. En las *cabeceras TCP* se añaden el puerto de destino y el de origen, que identifican respectivamente la aplicación a la que va dirigido el paquete en el *host* de destino, y la que lo generó en el *host* de origen.

El protocolo *TCP* ofrece a la aplicación – que se encuentra en el nivel inmediatamente superior – un servicio orientado a conexión. Esto significa que *TCP* crea una secuencia de bytes para el envío y otra para la recepción, de tal forma que la aplicación podrá escribir en la primera y leer de la segunda mientras permanezca abierta la conexión. Teniendo en cuenta que el protocolo *TCP* envía y recibe paquetes, es evidente que tiene que realizar la operación de *segmentación* – descomponer la secuencia de datos en paquetes – en el envío, y *reensamblado* en la recepción – unir los paquetes recibidos para formar una secuencia - .

En el nivel de ***Aplicación***, se encontrará el protocolo de comunicaciones de alto nivel, directamente relacionado con la aplicación de usuario. Algunos de los protocolos estándar de Aplicación son *HTTP* – HyperText Transfer Protocol -, *FTP* – File Transfer Protocol -, *SMTP* – Simple Mail Transfer Protocol - , etc.

Así, por ejemplo, si usamos un navegador para conectarnos a un servidor *HTTP*, tanto el navegador en el *host* local como el servicio *HTTP* en el servidor implementan el protocolo *HTTP* en el nivel de *aplicación*.

Es en el nivel de *aplicación*, precisamente en el que nosotros desarrollaremos las comunicaciones de red para nuestras aplicaciones, usando los servicios proporcionados por los niveles inferiores, principalmente *TCP*.

En las redes *TCP/IP* se usa el modelo ***Cliente – Servidor*** que distingue los dos extremos de una comunicación. De esta forma el servidor es un programa o servicio de red que se encuentra permanentemente en espera de conexiones desde los clientes. El cliente es siempre el que inicia una conversación, realizando una conexión con el servidor. Cuando esto ocurre, el servidor inicia la conversación con el cliente y a partir de ese momento ambos mantienen la conexión como iguales hasta que uno de ellos la cierra. Normalmente el servidor, cuando inicia una conexión con un cliente, se sigue manteniendo a la escucha para atender conexiones de otros clientes.

El proceso servidor se enlaza a un puerto específico, que será conocido por los clientes con el fin de poder establecer la conexión con dicho puerto. Los puertos más bajos – desde el 0 hasta el 1023 – están reservados para los servicios estándar. Así, por ejemplo, cuando un cliente web se conecta a un servidor, lo hace poniendo como dirección IP de destino la del *host servidor*, y como puerto de destino, el puerto 80 – que es el puerto predeterminado para el protocolo *HTTP* -. Los procesos clientes, por el contrario, se enlazan a un puerto asignado automáticamente por el sistema.

2. Manejo de URL's.

Una ***URL*** – Uniform Resource Locator – es una cadena de texto con un formato determinado que sirve para identificar un recurso – fichero, imagen, página web, etc – que se encuentra en un servidor.

Las partes más importantes de una *URL* son:

- Protocolo: identifica el protocolo usado por el servicio que proporciona el recurso. Ejemplos: `http`, `https`, `ftp`, `smtp` ...
- Host: el nombre o dirección del host servidor. Ejemplo: `www.google.com`
- Puerto(opcional): indica el puerto en el que se encuentra el servicio. Si no se especifica se toma el puerto por defecto para el protocolo indicado. Se separa del host mediante el carácter ":".
- Recurso (opcional): Indica el recurso que queremos obtener del servidor. Se puede indicar una ruta dentro del servidor. En ese caso se usa "/" como separador de ruta. Ejemplos: `index.html`, `imagenes/logo.gif`, ...
- Otras opciones, como parámetros, anclas, etc. Se especifican al final y dependen del protocolo usado. Ejemplos: `#ancla` , `?parametro=valor`, ...

Un ejemplo de URL puede ser

```
http://docs.oracle.com/javase/7/docs/api/java/net/URL.html
```

donde el protocolo es **`http`**, el host servidor es **`docs.oracle.com`**, y el recurso es **`javase/7/docs/api/java/net/URL.html`**.

En la API java.net existe la clase ***URL***, que además de encapsular una dirección *URL* con todas sus partes, proporciona métodos para realizar la conexión con el servidor para obtener el recurso especificado.

Para crear un objeto *URL* podemos usar cualquiera de sus constructores, de los cuales, el más sencillo toma como parámetro una cadena con la dirección *URL*. Existen otros constructores para crear una *URL* a partir de sus partes, o una *URL* relativa a otra, etc.

Una vez creado el objeto *URL*, podemos conectarnos al servidor y obtener el recurso especificado mediante el método *openStream()*, que devuelve un objeto de tipo *InputStream*, del cual podemos leer secuencias de bytes mientras permanezca abierto.

El siguiente fragmento de código muestra un método que lee el texto – código fuente *HTML* – devuelto por el servidor cuando se le solicita la *URL* pasada como parámetro.

```
  public void leerHTML(String cadenaURL) {
    URL url = null;
    try {
        // Creamos una URL a partir del texto introducido
        url = new URL(cadenaURL);
    } catch (MalformedURLException e) {
        salida.setText("URL incorrecta");
        return;
    }
    salida.setText("");
    BufferedReader entrada = null;
    try {
        // Obtenemos una secuencia de entrada de caracteres
        // para leer la respuesta del servidor
        entrada = new BufferedReader(new
                      InputStreamReader(url.openStream()));
    } catch (IOException e) {
        salida.setText("Error al abrir " + url.getHost());
    }
    String linea;
    try {
        // Mientras el servidor mantenga la conexion abierta..
        while ((linea = entrada.readLine()) !=null)
            salida.append(linea + "\n");
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
    try {
        // cerramos la secuencia de entrada
        entrada.close();
    } catch (IOException e) {
        System.err.println(e.getMessage());
    }
}
```

Ésta es la forma más directa de obtener un recurso del servidor cuando no se necesita mayor interacción.

También disponemos de otra técnica más elaborada para conectarnos a un servidor mediante la clase *URL*. Usando el método *openConnection()* obtenemos un objeto de tipo *URLConnection*, que nos permite establecer la conexión con el servidor, obtener una secuencia de entrada para leer los datos recibidos y una secuencia de salida para enviar datos al servidor.

El método *getInputStream()* de la clase *URLConnection* nos devuelve un objeto de tipo *InputStream*, a través del cuál podemos leer los datos recibidos, de la misma forma que lo hacíamos para leer directamente del objeto *URL*.

Además, el método *getOutputStream()* devuelve un objeto de tipo *OutputStream* que permite escribir secuencias binarias – o de texto, si realizamos las conversiones adecuadas -, que serán enviadas al servidor. Para hacer esto, debemos habilitar previamente la escritura en el objeto *URLConnection*, mediante una llamada al método *setDoOutput()*.

El siguiente fragmento de código muestra cómo conectarse a un recurso web usando el método *POST* y enviando un parámetro en la petición.

```
URL url = null;
try {
    url = new URL("http://localhost:8080/Web/cabeceras.jsp");
} catch (Exception e) {}
URLConnection conexion;
try {
    conexion = url.openConnection();
} catch (IOException e) {
    System.err.println(e.getMessage());
    return;
}
// Habilitamos el envio de datos
conexion.setDoOutput(true);
// Establecemos el metodo HTTP como POST
conexion.setRequestProperty("METHOD", "POST");
try {
    BufferedWriter salida = new BufferedWriter(new
    OutputStreamWriter(conexion.getOutputStream()));
```

```
            salida.newLine();
            salida.newLine();
            String parametros = "parametro=valor";
            salida.write(parametros);
            salida.close();
            BufferedReader entrada = new BufferedReader(new
            InputStreamReader(conexion.getInputStream()));
            String linea = null;
            while ((linea = entrada.readLine())!=null) {
                System.out.println(linea);
            }
            entrada.close();
        } catch (Exception e) {
            System.err.println(e.getMessage());
```

3. Tratamiento de sockets en Java. Diseño de programas clientes y servidores.

Un ***socket*** representa un extremo de la conexión entre dos programas en la red. Ofrece un flujo de entrada (*InputStream*) y un flujo de salida (*OutputStream*) que se usan para la recepción y el envío de datos respectivamente. Dichos flujos son binarios (flujos de bytes), aunque se pueden encapsular en objetos de lectura y escritura de cadenas de caracteres (*BufferedReader* y *PrintWriter*, por ejemplo).

La escritura de un programa cliente consta de los siguientes pasos:

- Creación de un *socket* y establecimiento de la conexión mediante **`new Socket(servidor,puerto)`**. El primer parámetro es una cadena que contendrá la dirección del *host servidor* – ya sea mediante un nombre de dominio o una dirección IP -, y el segundo parámetro será un número entero que indicará el *puerto TCP* en el que se encuentra el servicio al que nos queremos conectar.

- Obtención de los flujos de entrada y salida mediante **`getInputStream()`** y **`getOutputStream()`** respectivamente.

- Comunicación con el servidor mediante lecturas y escrituras a través de los flujos obtenidos. La secuencia de las lecturas y escrituras y la semántica de los mensajes dependen del *protocolo de aplicación*.

- Cierre de los flujos de datos y del propio *socket*.

En el siguiente fragmento de código se muestra la secuencia anterior, omitiendo los bloques de control de excepciones, para simplificar:

```
        String servidor = "localhost";
        int puerto =  7;
        …
        // Establecemos la conexión con el servidor en el puerto indicado
        Socket conexión = new Socket(servidor,puerto);
        …
        // Obtenemos las secuencias de entrada y de salida de la conexión
       BufferedReader entrada = new BufferedReader(new
InputStreamReader(conexion.getInputStream()));
        PrintWriter salida = new
PrintWriter(conexion.getOutputStream(),true);
        …
        // Mantenemos conversacion con el servidor
        salida.println("mensaje");
        …
        String mensajeRecibido = entrada.readLine();
       …
        // Cerramos secuencias de entrada y salida y conexión
        salida.close();
        entrada.close();
        conexión.close();
```

La escritura de un programa servidor difiere de la anterior, principalmente, en la forma de realizar la conexión. El servidor debe registrarse en el puerto reservado para el servicio, y a partir de ese momento aceptará conexiones de clientes. Cada vez que se acepta una conexión, se obtiene ésta en forma de un *socket*, que ya se puede utilizar para mantener la conversación con el cliente:

- Creación de un *socket servidor* y enlace a un puerto mediante **`new ServerSocket(puertoServidor)`**.

- Aceptando conexiones de clientes y obteniendo *sockets cliente* para realizar la comunicación: **`socketCliente = socketServidor.accept()`**.

- Obtener las secuencias de entrada y salida de éste *socket* y mantener la comunicación con el cliente mediante lecturas y escrituras sobre dichas secuencias siguiendo las reglas definidas en el protocolo de aplicación.

- Cerrar secuencias y *socket* cliente.

- Cerrar el *socket* servidor.

Para aceptar múltiples conexiones simultáneas se crea un *hilo de ejecución* para manejar cada *socket cliente* obtenido mediante `socketServidor.accept()`.

El siguiente código muestra un servidor de *echo*:

```
package echo;

import java.net.*;
import java.io.*;

public class EchoServer {

    public static void main(String[] args) {
        // Conectamos el servicio al puerto 4000
        ServerSocket socketServidor = null;
        try {
            socketServidor = new ServerSocket(4000);
        } catch (IOException ex) {
            System.err.println(ex.getMessage());
            System.exit(-1);
        }
        // Aceptamos conexiones
        while (true) {
            try {
                Socket conexion = socketServidor.accept();
                // Creamos un nuevo subproceso para cada conexion de
cliente
                new ConexionCliente(conexion).start();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
```

```
        }
    }

    private static class ConexionCliente extends Thread {
        private Socket conexion;

        public ConexionCliente(Socket s) {
            conexion = s;
        }

        @Override
        public void run() {
            try {
                // Obtenemos las secuencias de entrada y salida de la
conexion
                BufferedReader entrada = new BufferedReader(new
InputStreamReader(conexion.getInputStream()));
                PrintWriter salida = new
PrintWriter(conexion.getOutputStream(),true);
                // Mientras la entrada de la conexion permanezca abierta
                // Leemos mensaje y lo retransmitimos
                String linea;
                while ((linea = entrada.readLine()) != null) {
                    salida.println(linea);
                }
                // Cerramos la conexion
                conexion.close();
            } catch (IOException ex) {
                System.err.println(ex.getMessage());
            }
        }
    }
}
```