DESARROLLO DE SERVICIOS TELEMÁTICOS

Tema 1

3er. curso

Graduado en Informática

Mención de Tecnologías de la Información

Autora: Lidia Fuentes

Desarrollo de servicios clásicos de Internet

- Programación avanzada con sockets
- Desarrollo de Clientes y Servidores sobre UDP
- Desarrollo de Clientes sobre TCP
- Patrones de diseño para aplicaciones distribuidas

1.1. Introducción

- Objetivo
  - Diseñar e implementar aplicaciones/servicios telemáticos sobre la red Internet
- Requisitos
  1. Utilizar un mecanismo de comunicación entre procesos independientes (distribuidos o no)
  2. Diseñar y/o implementar un protocolo de nivel de aplicación
  3. Definir el identificador del servicio/aplicación
  - **Esquema pizarra**

Mecanismos de comunicación

- Procesos concurrentes
- Mecanismos de comunicación entre procesos locales
  - *IPC (Inter-Process Communication)*
- Comunicación entre procesos remotos
- **Esquema pizarra**

Procesos concurrentes

- Hebras (*Threads*)
  - Estándar POSIX (lenguaje “C”/Unix/Linux)
  - Implementación en Java
- Características
  - Permite delegar tareas en diferentes procesos
  - Ejecución concurrente de tareas en una misma máquina
  - Permite que un proceso servidor pueda gestionar varias peticiones concurrentemente
- Desventajas
  - Cohesión y acoplamiento de las tareas
  - No funcionan igual para todos los S.O o Java
  - Poco control del orden de ejecución de las tareas

Mecanismos de comunicación entre procesos locales (IPC)

- Estándar POSIX (lenguaje “C”/Unix/Linux)
  - Paso de mensajes (pipes, FIFOS, colas de mensajes)
  - Sincronización (semáforos, mutex y variables de condición)
  - Memoria compartida (con semáforos)
    - Para grandes cantidades de datos
- IPC en Java
  - Soporte pobre en Java
  - Bibliotecas de código abierto
    - CLIPC (http://sourceforge.net/projects/clipc)
  - Uso de IPCs en “C” mediante JNI

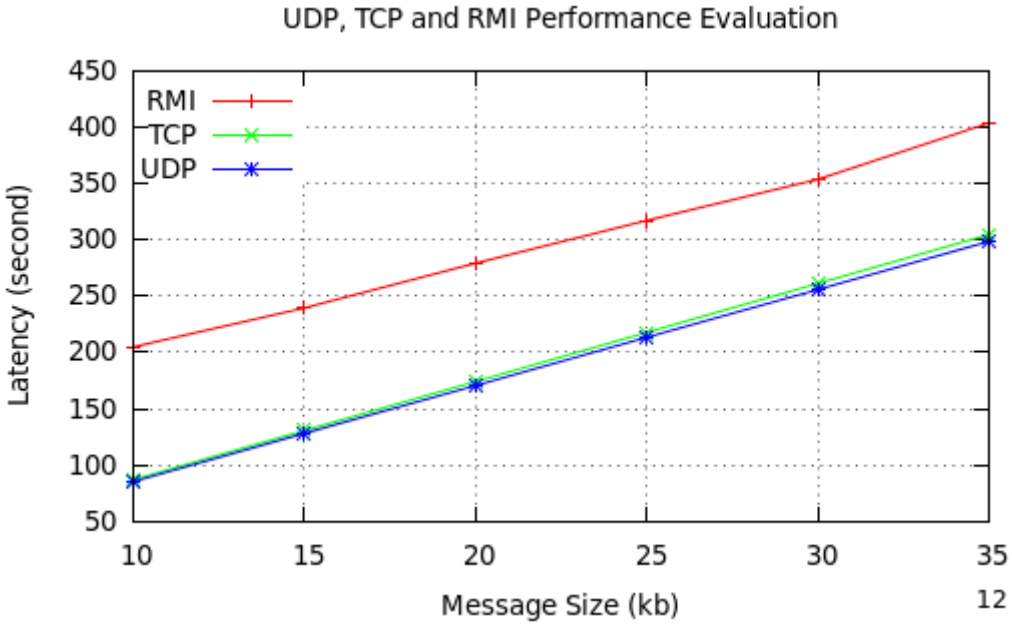
Comunicación entre procesos remotos

- Pila de protocolos de Internet (TCP/IP)
  - Protocolos de transporte
    - Orientado a la conexión (TCP)
    - No orientado a la conexión (UDP)
- Dispositivos
  - Máquinas servidores, equipos de sobremesa, …
  - Equipos portables
    - Tabletas, teléfonos móviles, sensores, etc.
- **Esquema pizarra**

Comunicación entre procesos remotos

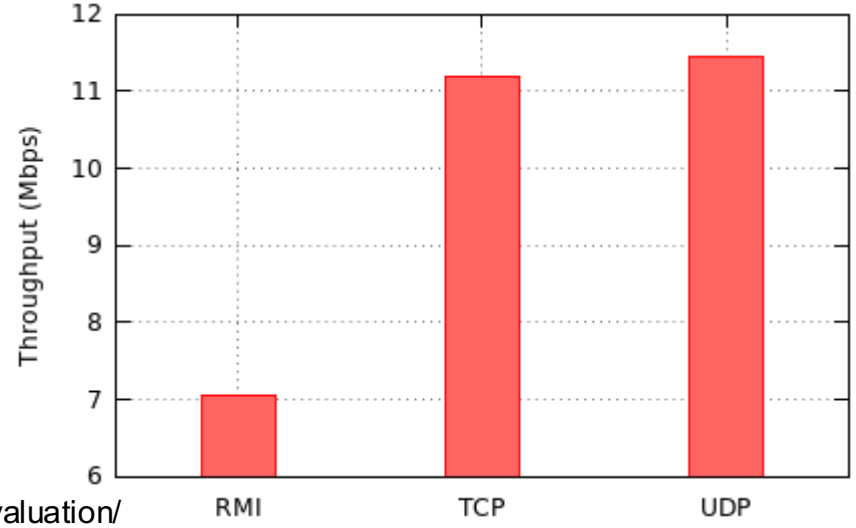
- Sockets (API estándar en cualquier lenguaje de programación)
  - Comunicación asíncrona
  - Programación de bajo nivel, pero lo más eficiente
  - El programador es responsable de codificar y decodificar los datos (ristra de bytes)
- Llamada a procedimiento remoto (RPC, o RMI/Java)
  - Comunicación síncrona
  - Programación de más alto nivel, menos eficiente y portable
  - Hay que conocer la ubicación del procedimiento (método) remoto
- **Esquema pizarra**

RMI vs Sockets TCP UDP



Latencia: tiempo de transmisión de paquetes

Rendimiento: Tamaño paquetes recibidos por unidad de tiempo



http://lycog.com/performance-evaluation/tcp-udp-rmi-performance-evaluation/

Comunicación entre procesos remotos

- Servicios Web
  - Sobre HTTP (Servlets, páginas JSP, etc.)
  - Invocación de servicios remotos (SOAP) en un servidor Web
- Plataformas distribuidas (middleware)
  - CORBA, CCM/CORBA, Java EE, OSGi, JINI, JSM, etc.
  - No necesario conocer ubicación procesos locales/remotos
- Versiones ligeras para dispositivos ligeros
  - Java ME
  - Android
  - OSGi
- **Esquema pizarra**

Diseñar y/o implementar un protocolo de nivel de aplicación

- Servicios estándar de Internet
  - Descritos en RFCs
  - Adherirse al estándar
- Servicios no estándar
- Elegir un mecanismo de comunicación
  - Tema 1: sockets
  - Tema 2: servicios web

Servicios estándar de Internet

Servicio	ICMP	UDP	TCP
ping	■		
traceroute	■	■	
BOOTP		■	
DHCP		■	
NTP		■	
TFTP		■	
SNMP		■	
SMTP			■
Telnet			■
FTP			■
HTTP			■
NNTP			■
DNS		■	■
NFS		■	■
RMI			■

Definir el identificador del servicio/aplicación

- Puertos de transporte (asignación según IANA) short de 16 bytes

Descripción	Rango de puerto
Puerto del sistema • Solicitud de puerto efímero	0
Puertos reservados • Conocidos y asignados a servicios estándar clásicos • Asignados por IANA	1 – 1023
Puertos registrados • Registrados por servicios no clásicos (ej: HTTPS, CORBA, RMI, etc.) • No deben ser usados sin registrarlos en IANA	1024 – 49151
Puertos privados o dinámicos	49151 – 65535

IANA: Internet Assigned Numbers Authority

1.2. Requisitos de los servicios telemáticos

- Dos protocolos de transporte: TCP y UDP
- Programación con sockets UDP
  - Características del servicio UDP
  - Diseño de servicios sobre UDP
- Programación con sockets TCP
  - Características del servicio TCP
  - Diseño de servicios sobre TCP
- Decidir entre TCP y UDP

Características del servicio UDP

- No orientado a la conexión (servicio datagrama)
- Ofrece servicio no fiable (hay que codificar a nivel de aplicación):
  - No. de secuencia de mensaje (si hay varios, ej: TFTP)
  - Confirmaciones (varios mensajes, ej: TFTP)
  - Temporizador de retransmisiones
  - Número máximo de retransmisiones
  - Identificador de conexión (o transacción) para rechazar mensajes no pertenecientes a la transacción actual

Diseño de servicios sobre UDP

- Codificación de mensajes
  - Paquetes de longitud fija
  - Codificación binaria
- Lectura/Escritura de mensajes
  - Hay una correspondencia entre envío -> recepción de mensajes
  - Si se envía un datagrama de tamaño mayor que el buffer de UDP, entonces se trunca o overflow
  - Lectura bloqueante + temporizador + retransmisión
- Fin del servicio
  - Mensaje FIN_SERVICIO
  - Longitud mensaje < Longitud fija
- Servidores
  - Iterativos para aplicaciones estándar (concurrentes, menos frecuentes)

Características del servicio TCP

- Orientado a la conexión (establecer una conexión, intercambio de datos, liberación de la conexión)
- Full-duplex: sólo necesito una conexión tanto para el envío como para la recepción
- Entrada/salida basada en buffers
  - Por defecto no se usa esta característica activando el flag PSH en la cabecera de los segmentos TCP de datos
  - Es posible modificar este comportamiento desde código
- Envío de datos fiables (no hay que codificar retransmisiones de mensajes)
- Envío de datos basado en flujo de bytes
  - Para TCP los datos no tienen tipo (ej: int, char, etc.)
  - Conversión explícita e implícita de tipos de datos específicos a una lista de bytes

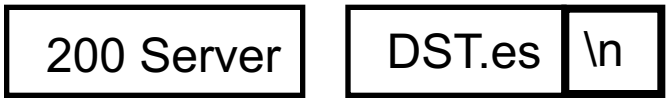
Diseño de servicios sobre TCP

- Codificación de mensajes
  - Mensajes de longitud variable
  - Codificación textual
  - Campo tipo de mensaje
    - 3 o más caracteres
    - Ej: En FTP: USER, RETR, STOR, etc.
  - Límite de mensajes
    - Necesario definir un carácter de FIN_MENSAJE (“\r\n”)
  - Protocolos estándar
    - [CCC|CCCC][<SP><parametros>]<CRLF>
    - **Ejemplo en la pizarra**

Diseño de servicios sobre TCP

- Lectura/Escritura de mensajes
  - Puedo leer varios mensajes en una única operación de lectura

  - Puede llegarme un mensaje en varias operaciones de lectura



  - Un ejemplo con todos los casos

Diseño de servicios sobre TCP

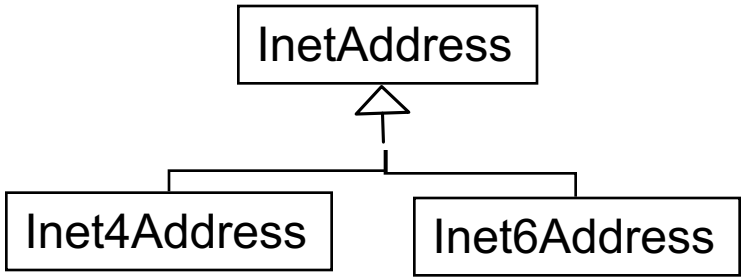
- Fin del servicio
  - Cierre de conexión finaliza la comunicación
  - Mensaje FIN_SERVICIO
- Servidores
  - Iterativos
  - Concurrentes (lo normal)
  - Superservidores
    - Atienden diferentes servicios de forma concurrente

Decidir entre TCP y UDP

- UDP vs TCP
  - Regla: Usar TCP a menos que tengas una buena razón para no hacerlo
- Razones para usar UDP
  - Tener que hacer *broadcast* o *multicast*
  - Envío de datos en tiempo real, y tolerantes a fallos (datos multimedia, de vídeo)
  - Queremos implementar un servicio con transacciones de pocos mensajes (Ej: envío+recepción)

1.3. Programación de servicios con Sockets

- Sockets de Java
- Direcciones IP
  - Clase InetAddress
  - Ejemplo IPv4: `host4.es/150.214.108.54`
  - Ejemplo IPv6: `host6.es/1230::340:1b54:75a4`



Nombre host/Dirección(es) IP

Clase InetAddr

- Representa una dirección IP y tiene los siguientes métodos:

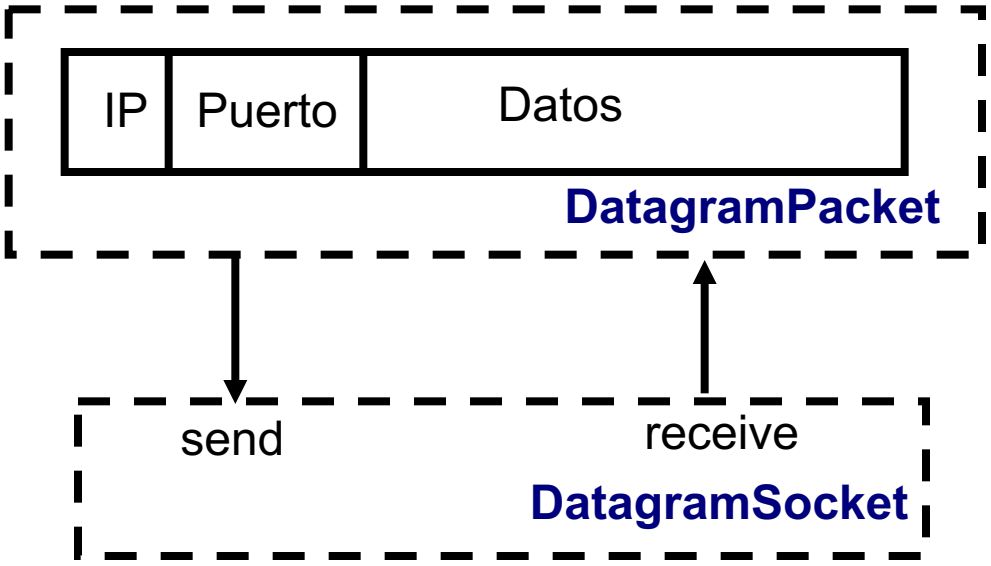
```
static InetAddress getLocalHost() throws
   UnknownHostException
   /* Dirección del host local */
/* nombre/direccion IP Ej: lff.lcc.uma.es/150.214.108.4 */
static synchronized InetAddress getByName(String host)
   /* Devuelve la dirección IP del host */
static synchronized InetAddress[] getAllByName(String host)
   /* Devuelve todas las direcciones IP de un host */
byte[] getAddress()
   /* Devuelve la dirección IP en un array de 4 bytes */
String getHostName()
   /* Devuelve el nombre del host del objeto */
String getHostAddress()
  /* Devuelve la IP del objeto "150.214.108.4" */
```

No son estáticos

```
// Obtiene el par nombre/dir. IP del host local y un host

import java.net.*;

public class GetHost {
  public static void main(String args[])
                                        throws Exception {
    System.out.println("Host local :"+
                                    InetAddress.getLocalHost());
    if (args.length != 1) {
      System.err.println("GetHost <host>");
      System.exit(1);
    }
    InetAddress ia=InetAddress.getByName(args[0]);
    System.out.println("Host pedido: "+ia);

  }
}
SALIDA:
Host local : lff/150.214.108.4
Host pedido: gisum/150.214.108.54
```

Sockets UDP

- Clase DatagramPacket
  - Modela el contenido de un datagrama UDP
- Clase DatagramSocket
  - Define un socket UDP



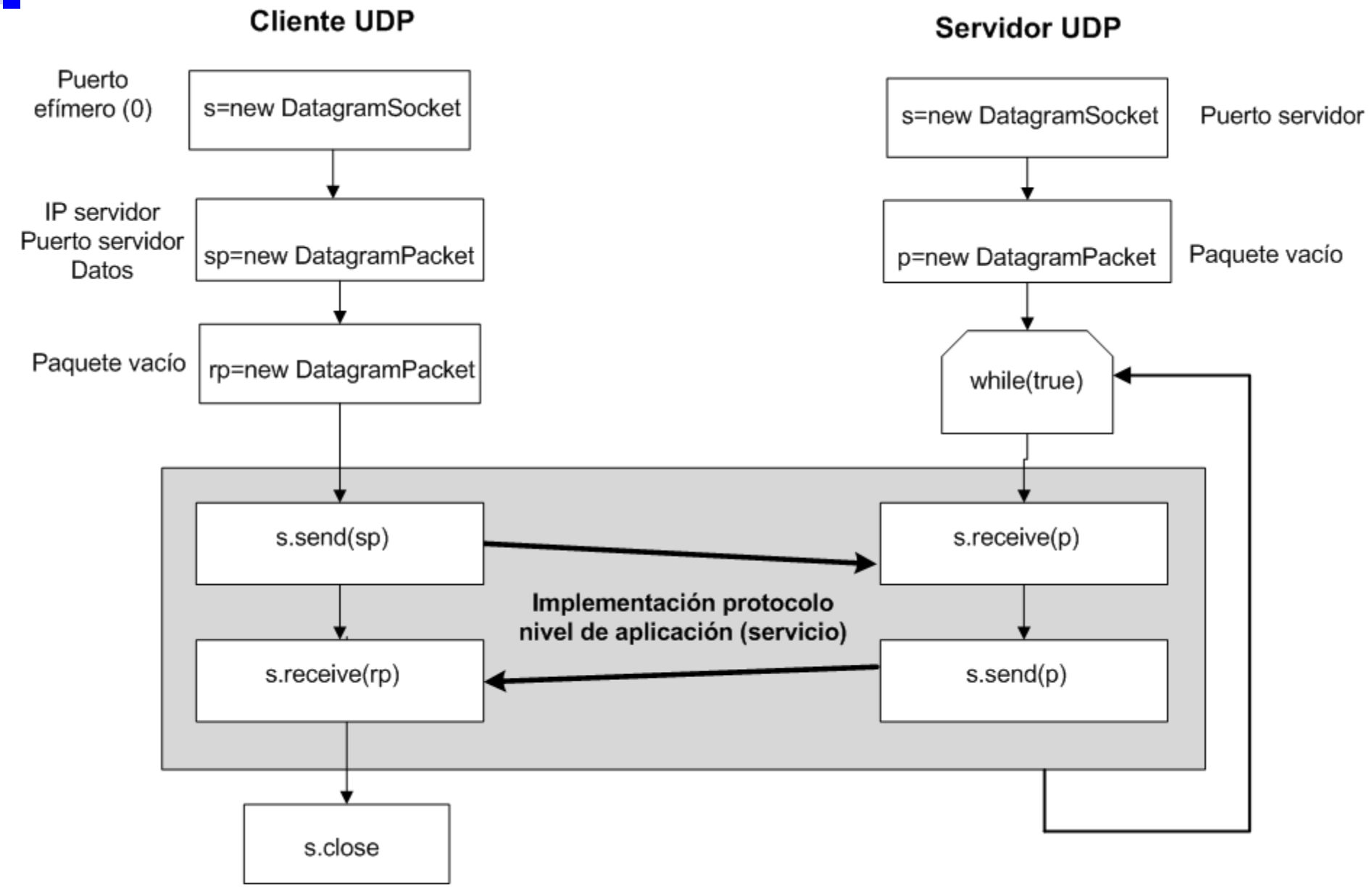
Clase DatagramPacket

```
DatagramPacket(byte buffer[],int n)
   /* datagrama UDP que se recibe a través de un socket UDP
   */
DatagramPacket(byte buffer[],int n,InetAddress host,int
   puerto)
   /* datagrama UDP que se envía indicando dirección y
   puerto destino */
InetAddress getAddress()
int getPort()
   /* devuelven la dirección y el puerto de la última
   lectura */
byte[] getData()
int getLength()
   /* devuelven los últimos datos leídos y la cantidad */
```

Se usa el tipo byte, no String !!

Clase DatagramSocket

```
DatagramSocket()throws SocketException
DatagramSocket(int puerto) throws SocketException
   /* Crea un socket UDP asignándole opcionalmente un
   puerto */
void send(DatagramPacket p) throws IOException
synchronized void receive(DatagramPacket p) throws
   IOException
   /* envío y recepción de datagramas UDP */
int getLocalPort()
   /* devuelve el puerto efímero local */
synchronized void close()throws IOException
   /* cierra el socket UDP */
```


Cliente UDP
Servidor UDP
Puerto efímero (0)
s=new DatagramSocket
s=new DatagramSocket
Puerto servidor
IP servidor
Puerto servidor
Datos
sp=new DatagramPacket
p=new DatagramPacket
Paquete vacío
Paquete vacío
rp=new DatagramPacket
while(true)
s.send(sp)
s.receive(p)
Implementación protocolo
nivel de aplicación (servicio)
s.receive(rp)
s.send(p)
s.close

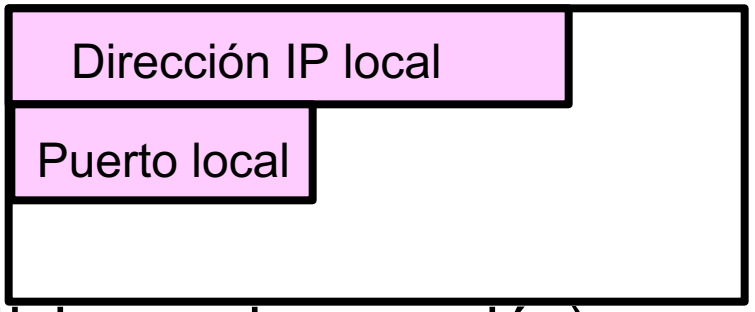
Servidor de Eco simple

```
import java.io.IOException;
import java.net.*;

public class UDPEchoServer {

  private static final int ECHOMAX = 255; // Tamagno maximo de los mensajes

  public static void main(String[] args) throws IOException {


    DatagramSocket socket = new DatagramSocket(6789);
    DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);

    while (true) {
      socket.receive(packet); // Recibe un datagrama del cliente
      System.out.println("IP cliente: " + packet.getAddress().getHostAddress() +
                         " Puerto cliente: " + packet.getPort());
      socket.send(packet); // Enviar el mismo datagrama al cliente
      packet.setLength(ECHOMAX); // Limpiar buffer
    }
}
}
```

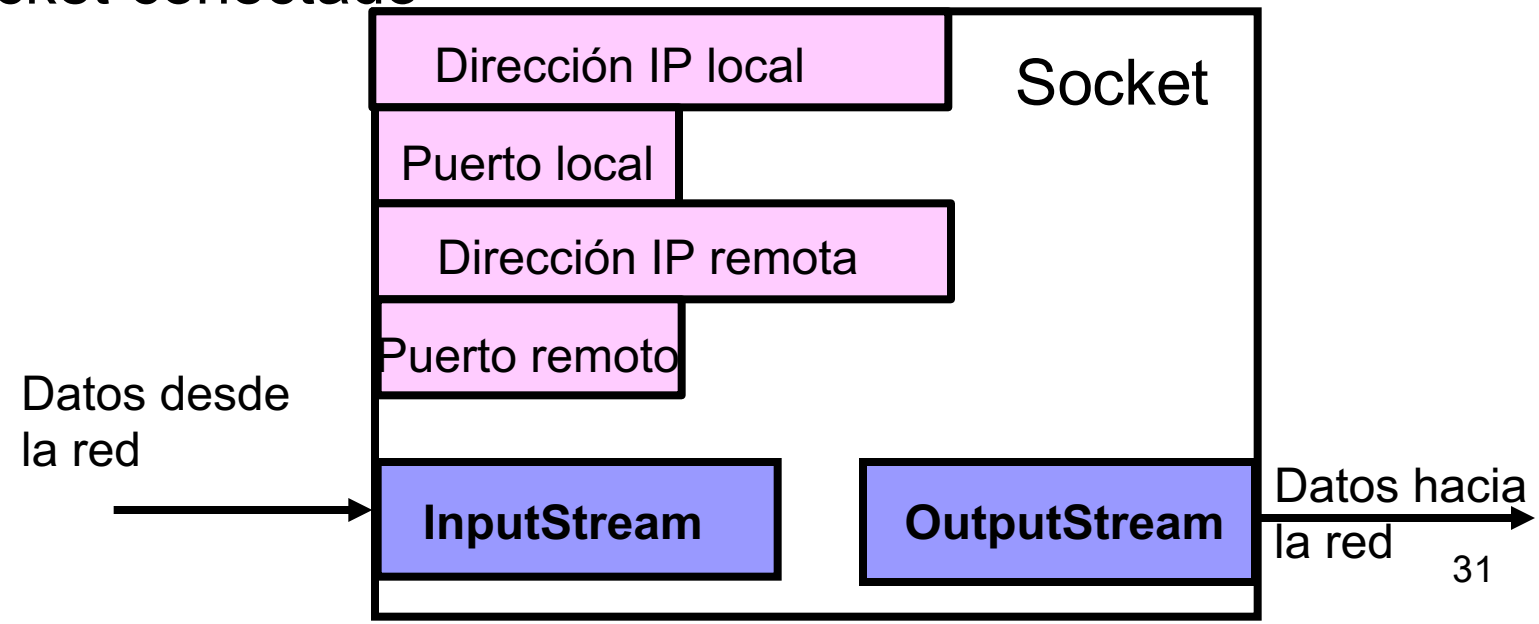
Cliente de Eco simple

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

public class UDPEchoClient {

  public static void main(String[] args) throws IOException {

      if ((args.length < 1) || (args.length > 2)) {
              throw new IllegalArgumentException("Parameter(s): <Server> [<Port>]");
            }
      InetAddress serverAddress = InetAddress.getByName(args[0]);  // IP Servidor

      int servPort = (args.length == 2) ? Integer.parseInt(args[1]) : 6789;
      BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));

      String mensaje=stdIn.readLine();
      byte[] bytesToSend = mensaje.getBytes();

      DatagramSocket socket = new DatagramSocket();
      DatagramPacket sendPacket = new DatagramPacket(bytesToSend, //Datagrama a enviar
                bytesToSend.length, serverAddress, servPort);
      socket.send(sendPacket);
      DatagramPacket receivePacket =                              //Datagrama a recibir
                new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);
      socket.receive(receivePacket); // Podria no llegar nunca el datagrama de ECO
      System.out.println("ECO:"+ new String(receivePacket.getData()));
      socket.close();
}}
```

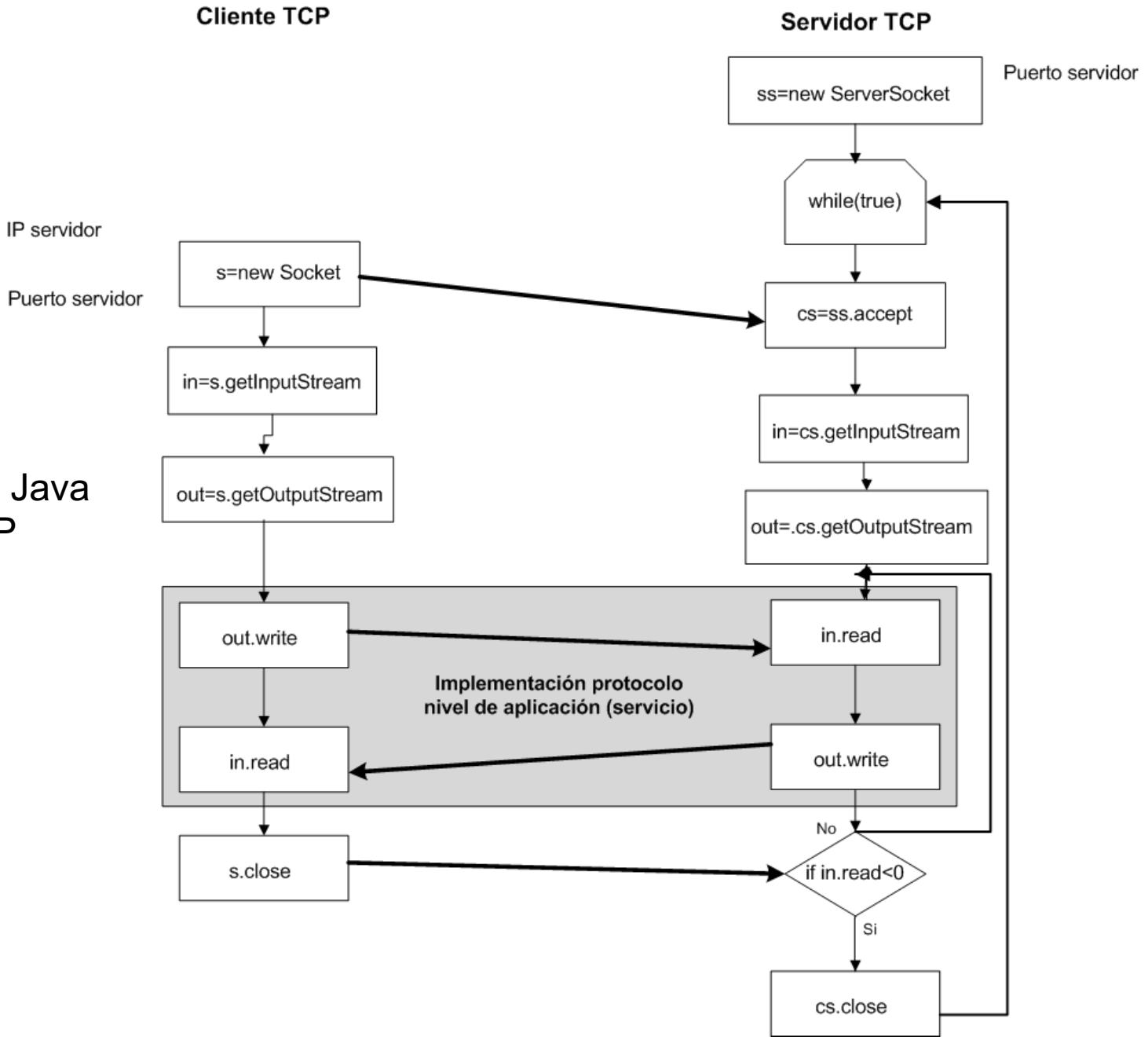
Sockets TCP



- ServerSocket(int puerto)
  - Socket pasivo (escucha peticiones de conexión)
- Socket()
  - Socket conectado



Sockets Java TCP




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

public class EchoClientTCP {
  static final int serverPort = 6543;
  public static void main(String[] args) throws IOException {
      try {
        InetAddress serverAddr = InetAddress.getByName();
        Socket sockfd = new Socket(serverAddr,serverPort);
        System.out.println("Conexión local"+serverAddr);

        BufferedReader in = new BufferedReader(
               new InputStreamReader(sockfd.getInputStream()));
        PrintWriter out = new PrintWriter(new BufferedWriter
           (new OutputStreamWriter(sockfd.getOutputStream())),
            true);
        BufferedReader stdIn = new BufferedReader(
                              new InputStreamReader(System.in));
        String userInput; //Entrada por teclado
```

```
      while ((userInput = stdIn.readLine()) != null) {
         if (userInput.equals(“.”)) break; // finaliza con el “.”
         out.println(userInput); //escribo socket
         System.out.println("echo: " + in.readLine()); //leo socket
      }
      out.close();
      in.close();
      stdIn.close();
      sockfd.close();
      } catch (UnknownHostException e) {
            System.err.println("Unknown: " + serverAddr);
            System.exit(1);
        }
      catch (IOException e) {
            System.err.println(“Error I/O for " + serverAddr);
            System.exit(1);
        }
   }
}
```

```
import java.io.*;
import java.net.*;
class EchoServerTCP {
  public static void main(String args[]) {
    String line;
     try {
         ServerSocket sockfd = new ServerSocket(6543);
         System.out.println("Inicio servidor "+sockfd);

         while (true) {
            Socket newsockfd = sockfd.accept();
            System.out.println("Nuevo cliente, socket "+newsockfd);

            BufferedReader in = new BufferedReader(new InputStreamReader
                                        (  newsockfd.getInputStream()));
            PrintWriter out = new PrintWriter(new BufferedWriter
                              (new OutputStreamWriter
                                  (newsockfd.getOutputStream())),true);
```

```
        boolean salir = false;
        while(!salir) {
            line = in.readLine(); //lectura socket cliente
            if (line!=null) {
                out.println(line);} //escritura socket cliente
            else {
                salir = true;
            } // cierre socket cliente
        }
        newsockfd.close();
    }
  } catch (Exception e) {
      e.printStackTrace();
  }
 }
}
```

1.4. Desarrollo de servidores UDP

- Temporizador de retransmisiones
- Retransmisión de paquetes y número máximo de retransmisiones
- Identificador de conexión (o transacción) para rechazar mensajes no pertenecientes a la transacción actual
- Mensajes codificados en binario de longitud fija
- Servidores concurrentes

Gestión de temporizadores

- get/setSoTimeout(int) => SO_TIMEOUT
  - Opción de la clase *DatagramSocket*
  - Temporizador que cuando expira lanza la excepción *SocketTimeoutException*
  - Uso en servicios UDP (clase *DatagramSocket*)
    - Temporizador de pérdida de paquetes
    - Se pone antes de una lectura

```
sockfd.send(dpout);
try {
  sockfd.setSoTimeout(2000); // espera 2 seg
  DatagramPacket dpin= new DatagramPacket
          (new byte[256], 256);
  sockfd.receive(dpin);
  …
} catch (SocketTimeoutException e){ // timeout
    System.out.println("timeout - retransmision");
    sockfd.send(dpout);
  }
```

Temporizadores y retransmisión

```
public class UDPEchoClientTimeout {

  // Temporizador de retransmision (ms)
  private static final int TIMEOUT = 3000;
  // Maximo numero de retransmisiones
  private static final int MAXTRIES = 5;

  public static void main(String[] args) throws IOException {

    InetAddress serverAddress = InetAddress.getLocalHost();
    String s = new String("Asignatura DST");
    byte[] bytesToSend = s.getBytes();

    DatagramSocket socket = new DatagramSocket();

    DatagramPacket sendPacket = new DatagramPacket(bytesToSend,
                        bytesToSend.length, serverAddress, 5678);

    DatagramPacket receivePacket = new DatagramPacket(
                    new byte[bytesToSend.length], bytesToSend.length);
```

```
    int tries = 0;
    boolean receivedResponse = false;
    do {
      socket.send(sendPacket);
      socket.setSoTimeout(TIMEOUT); // Temporizador para cada envio
      try {
        socket.receive(receivePacket);

        receivedResponse = true;
      } catch (SocketTimeoutException e) {  // Expiro el temporizador
        tries += 1;
        System.out.println("Timeout, "+(MAXTRIES - tries)+" mas");
      }
    } while ((!receivedResponse) && (tries < MAXTRIES));

    if (receivedResponse) {
      System.out.println("Received: " + new String(
                                          receivePacket.getData()));
      socket.setSoTimeout(0);
    } else {
      System.out.println("No hubo respuesta del servidor.");
    }
    socket.close();
  }
}
```

Transacciones

- Identificador de transacción (TID)
  - usado por cliente/servidor para descartar mensajes no correspondientes a la transacción actual
- Cliente/servidor descartará el mensaje si no contiene el TID correcto
- BOOTP y DHCP
  - número aleatorio elegido por el cliente (es parte del mensaje DHCP)
- TFTP
  - Se utiliza como TID=<puerto_cli,puerto_serv>
  - Si el servidor es concurrente, para cada cliente puede elegir un puerto efímero aleatorio

Transacciones

- La opción mas sencilla es chequear la IP
- Ejemplo para un cliente

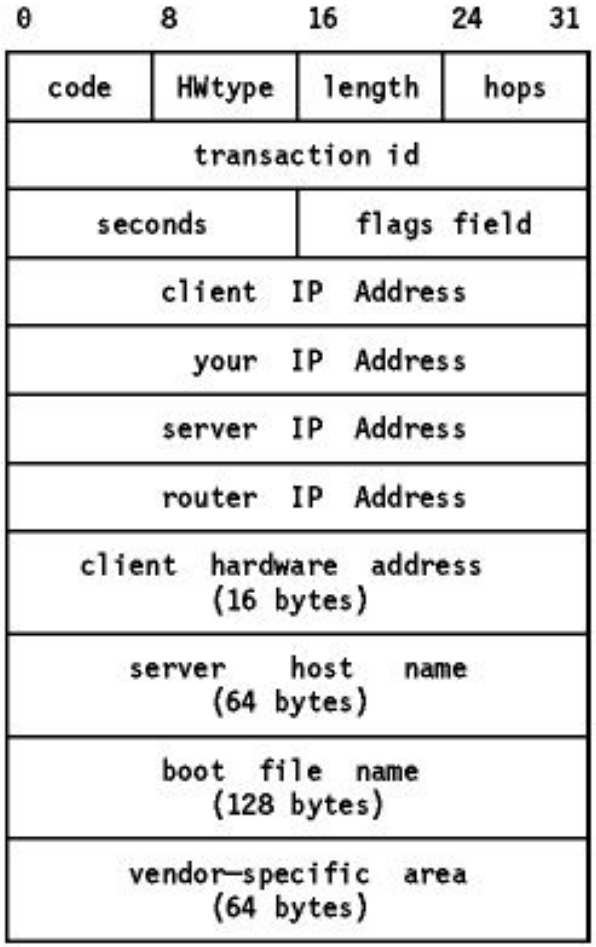
```
do {
   socket.send(sendPacket);
   try {
     socket.receive(receivePacket);

     if (!receivePacket.getAddress().equals(serverAddress))
       throw new IOException("Id. de transaccion incorrecto");
     }
     receivedResponse = true;
   } catch (IOException e) {
     tries += 1;
     System.out.println("Timeout, "+(MAXTRIES - tries)+" mas");
   }
 } while ((!receivedResponse) && (tries < MAXTRIES));
```

Codificación de mensajes

- Protocolos no estándar (propios)
  - Cliente y Servidor se ponen de acuerdo en la codificación
  - Serialización de objetos no recomendada
    - Se incluye más información de la necesaria
    - Es trabajosa de implementar para algunos mensajes
- Protocolos estándar
  - Mensajes de campos y longitud fija
  - Codificación binaria recomendada

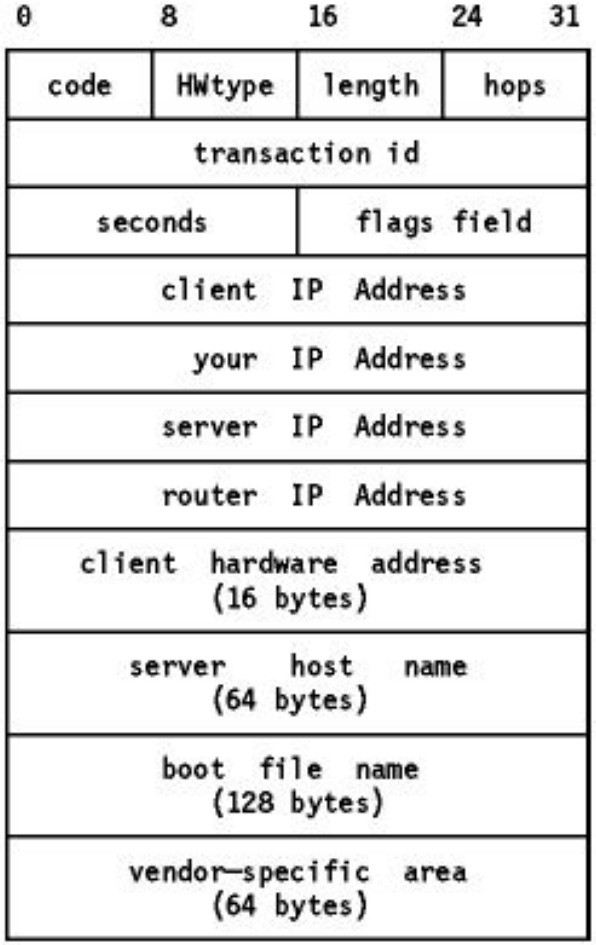
Mensaje BOOTP

```
public class BOOTPMsg {
  private static final int ADDR_LEN=4;

  private byte code;
  private byte Hwtype;
  private byte length;
  private byte hops;
  private int TransactionID;
  private short seconds;
  private short flags;
  private byte[] ciaddr=new byte[ADDR_LEN];
  private byte[] yiaddr=new byte[ADDR_LEN];
  private byte[] siaddr=new byte[ADDR_LEN];
  private byte[] giaddr=new byte[ADDR_LEN];
  private byte[] chaddr=new byte[16];
  private byte[] sname=new byte[64];
  private byte[] bootfile=new byte[128];
  private byte[] options=new byte[64];

  public void setCode(byte b) {this.code=b;}
  public byte getCode() {…}
  …
```



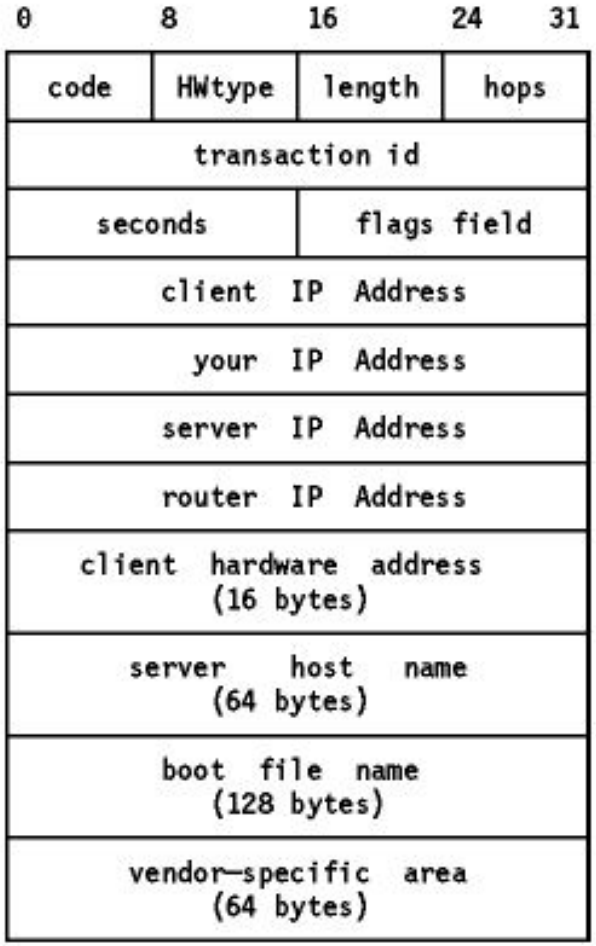
Codificación BOOTP

```
public byte[] encodeBOOTPMsg() throws IOException {
  ByteArrayOutputStream byteStream =
            new ByteArrayOutputStream();
  DataOutputStream out =
            new DataOutputStream(byteStream);

  out.writeByte(this.code);
  out.writeByte(this.Hwtype);
  out.writeByte(this.length);
  out.writeByte(this.hops);
  out.writeInt(this.TransactionID);
  out.writeShort(this.seconds);
  out.writeShort(this.flags);
  out.write(this.ciaddr, 0, ADDR_LEN);
  out.write(this.yiaddr, 0, ADDR_LEN);
  out.write(this.siaddr, 0, ADDR_LEN);
  out.write(this.giaddr, 0, ADDR_LEN);
  out.write(this.chaddr, 0, 16);
  …
  return byteStream.toByteArray();
}
```



Decodificación BOOTP

```
public void decodeBOOTPMsg(byte[] msg) throws IOException {
  ByteArrayInputStream byteStream =
            new ByteArrayInputStream(msg);
  DataInputStream in =
            new DataInputStream(byteStream);

  this.code = in.readByte();
  this.Hwtype = in.readByte();
  this.length = in.readByte();
  this.hops = in.readByte();
  this.TransactionID = in.readInt();
  this.seconds = in.readShort();
  this.flags = in.readShort();
  in.readFully(this.ciaddr,0,ADDR_LEN);
  in.readFully(this.yiaddr,0,ADDR_LEN);
  in.readFully(this.siaddr,0,ADDR_LEN);
  in.readFully(this.giaddr,0,ADDR_LEN);
  in.readFully(this.chaddr,0,16);  …
  …
}
}
```



Manejando bytes

- Inicializar array de bytes
  - Clase java.util.Arrays
    - Arrays.fill(byte[] a, byte val)
    - Arrays.fill(byte[] a, int fromIndex, int toIndex, byte val)
- Copia
  - System.arraycopy(arrayorigen, 0, arraydestino, 0, arrayorigen.length);
- String y array de bytes
  - Crear un string a partir de un array de bytes: String(byte [] b)
  - Obtener bytes de un String: s.getBytes()
- Entero a array de bytes
  - Con b=new ByteArrayOutputStream(), out=new DataInputStream(b), out.writeInt(i)
  - byte[] bytes = ByteBuffer.allocate(4).putInt(1500).array();

Opciones de *DatagramSocket*

- get/setReceiveBufferSize() – get/setSendBufferSize()
  - Cambia el tamaño del buffer de lectura/escritura
  - Un objeto Socket y DatagramSocket tiene buffers de envío y recepción
- get/setReuseAddress() => SO_REUSEADDR
  - Permite reutilizar la dirección (puerto)
  - En UDP para la misma IP se utiliza cuando el broadcast está habilitado
- get/setBroadcast()
  - Permite la difusión de datagramas
  - Por defecto debería estar habilitado

```
InetAddress broadAddr = InetAddress.getByName("255.255.255.255");
int broadPort = 3000;
ds = new DatagramSocket(broadPort, broadAddr);
ds.setBroadcast(true);
String s = new String("DATOS BROADCAST");
byte[] bytesToSend = s.getBytes();
DatagramPacket dp = new DatagramPacket(bytesToSend,
                    bytesToSend.length, broadAddr, broadPort);
ds.send(dp);  // Broadcast el mensaje de datos
```

Servidores UDP concurrentes

- La mayoría de los servicios estándar de UDP son iterativos
- ¿ Cuándo usaremos un servidor secuencial ?
  - Cuando el tiempo medio de respuesta (TR) sea pequeño
  - Depende del tiempo de servicio (TS), del tamaño de las colas (N) y de la frecuencia de acceso de los clientes (TCOMM)

```
TR = (N/2 + 1) * TS + TCOMM
```

  - Ej: El DAYTIME es el típico servicio secuencial.
- ¿ Cuándo usaremos servidores concurrentes ?
  - Cuando el tiempo medio de respuesta (TR) es grande o desconocido a priori
  - Cuando el tiempo de servicio de una petición (TS) es muy variable (Ej.: TFTP, comandos get y put)
  - Cuando la demanda de acceso sea elevada

Concurrencia y puertos UDP

- Problema
  - No puede reutilizarse el mismo socket para todos los clientes
- Solución
  - Se debe crear un socket para atender a cada uno de los clientes
- Se utilizará la concurrencia ligera (hebras o *Threads*) que proporciona Java

```
public class ConcurrentServerUDP {

  public static void main(String[] args) throws IOException {
    DatagramSocket ds = null;
    try {
        InetAddress localAddr = InetAddress.getByName("localhost");
        int wellKnownPort = 3000; // Puerto conocido del servidor
        ds = new DatagramSocket(wellKnownPort, localAddr);
        byte[] buffer = new byte[2048];
        DatagramPacket datagram = new DatagramPacket(buffer,
                                                     buffer.length);
        while (true) {
           ds.receive(datagram);
           System.out.println("Nueva peticion de servicio");
           // Inicio de una hebra para la peticion actual
           (new ServerUDPImpl(datagram)).start();
        }
     } catch (IOException e) {
         System.err.println("Error E/S en: " + e.getMessage());
         finally {
           if (ds != null)
             ds.close();
         }
     }
}
```

```
public class ConcurrentServerUDP {

  public static void main(String[] args) throws IOException {
    DatagramSocket ds = null;
    try {
        InetAddress localAddr = InetAddress.getByName("localhost");
        int wellKnownPort = 3000;
        ds = new DatagramSocket(wellKnownPort, localAddr);
        byte[] buffer = new byte[2048];
        DatagramPacket datagram = new DatagramPacket(buffer,
buffer.length);
        while (true) {
           ds.receive(datagram);
           System.out.println("Nueva peticion de servicio");
           // Inicio de una hebra para la peticion actual
           (new ServerUDPImpl(datagram)).start();
        }
     } catch (IOException e) {
         System.err.println("Error E/S en: " + e.getMessage());
         finally {
           if (ds != null)
             ds.close();
         }
     }
}
```

```
public class ConcurrentSe

  public static void mai
    DatagramSocket ds = n
    try {
        InetAddress loca
        int wellKnownPor
        ds = new Datagra
        byte[] buffer = n
        DatagramPacket da
buffer.length);
        while (true) {
            ds.receive(da
            System.out.pri
            // Inicio de u
            (new ServerUDI
        }
    } catch (IOExceptio
        System.err.print
        finally {
          if (ds != nul
            ds.close();
        }
    }
}
```

```
class ServerUDPImpl extends Thread {
  private byte[] datos;
  private InetAddress address;
  private int port;


  public ServerUDPImpl(DatagramPacket d) {
    this.datos=d.getData();
    this.address=d.getAddress();
    this.port = d.getPort();
  }

  public void run() {
    try {
      DatagramSocket ds = new
DatagramSocket(); // Nuevo socket y puerto
      DatagramPacket sd = new DatagramPacket
          (datos,datos.length,address,port);
      ds.send(sd);
      //DENTRO DE BUCLE SEGÚN IMPLEMENTACION
      //DEL PROTOCOLO
      ds.close();
    } catch (Exception e) {}
  }
}
```

3.2.1. Servidor concurrente UDP

Asignación de nuevo puerto en el proceso servidor hijo

```
servidor() {
  s=socket(…)
  bind(s, …); //asigna puerto=3000
  for(;;) {
    recvfrom(s,<pet_serv>,<cli_addr>, …);
    if ((cpid=fork()) == 0) { // hijo
      close(s);
      execl("/…/hijo",hijo,<cli_addr>,
                                  < pet_serv>,..);
      // <cli_addr> en un string
      // usar inet_ntop() para la IP y
      // ntohs() para puerto
     exit(0);
    } /* if */ …
  } /* for */
…
} /* servidor */
```

```
cliente() {
  s=socket(…)
  bind(s, …); //asigna puerto=5000
  …
  sendto(s,<pet_serv>,
     <serv_addr>, …); // puerto 3000
  recvfrom(s, ….,<serv_addr>,…);
     // puerto 3500
  sendto(s, …,<serv_addr>,…);
     // puerto 3500
….
} /* cliente */
```

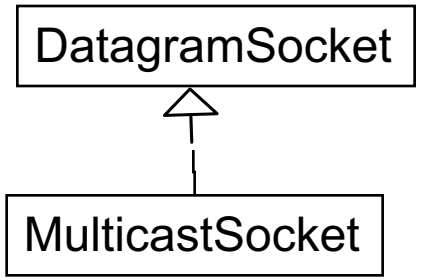
```
hijo() {
   sh=socket();
   bind(sh, …); // puerto = 3500
   sendto(sh, …, <cli_addr>,…); // puerto 5000
   recvfrom(sh,…,<cli_addr>,…); …;close(sh);
} /* hijo */
```

Sockets Multicast

- Los grupos multicast se identifican por una dirección multicast (clase D) y por el puerto al cuál está asociado el socket que quiere participar en el grupo.
- Todos los participantes de un grupo multicast tienen que poner la misma dirección y el mismo puerto
- Envía un datagrama al grupo multicast utilizando como TTL el establecido para este socket y no el valor por defecto para cualquier socket.
- El TTL define el alcance del multicast

Clase MulticastSocket

```
MulticastSocket(int)
     /* Crea un socket multicast y lo liga a un puerto
   determinado que será el representante del grupo (donde todos
   van a leer y a escribir). */
InetAddress getInterface()
setInterface(InetAddress)
     /* Devuelve/establece la dirección IP asignada al interfaz
   de red utilizado para enviar los datagramas multicast. */
int getTimeToLive()
setTimeToLive(int ttl)
     /* Establece el TTL de los datagramas, o sea que se puede
   determinar el ámbito hasta donde van a alcanzar los datagramas
   a la hora de preguntar si un determinado host tiene procesos
   escuchando en el puerto multicast establecido. */
joinGroup(InetAddress)
leaveGroup(InetAddress)
     // Unirse o dejar un grupo multicast
send(DatagramPacket, byte)
receive(DatagramPacket)
```



Ejemplo

- Creación:

```
MulticastSocket socket = new MulticastSocket(4446);
InetAddress group= InetAddress.getByName("230.0.0.1");
socket.joinGroup(group);
...
```

- Recepción

```
packet = new DatagramPacket(buf, buf.length);
socket.receive(packet);
```

- Envío

```
InetAddress group= InetAddress.getByName("230.0.0.1");
DatagramPacket packet = new DatagramPacket(buf,
buf.length, group, 4446);
socket.send(packet);
```

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

class MServer {
   public static void main(String args[]) throws Exception {


      // unirse a un grupo multicast y enviar hola
      byte[] msg = {'H', 'e', 'l', 'l', 'o'};

      /* Dirección multicast de clase "D". Si la definimos en
      el fichero /etc/hosts podremos utilizar un nombre */
      InetAddress group = InetAddress.getByName("228.5.6.7");
      System.out.println(group);

      /* Creo un socket multicast y lo asocio al puerto
        representante del grupo */
      MulticastSocket s = new MulticastSocket(6789);
```

```
    // Me uno al grupo identificado por 228.5.6.7 y puerto 6789
      s.joinGroup(group);

      // Envío el hello
      DatagramPacket hi = new DatagramPacket
                                  (msg, msg.length,group, 6789);
      s.send(hi);

      // leo las respuestas de los demás, incluida la mía !
      byte[] buf = new byte[1000];
      // tengo que hacer tantas lecturas como participantes
      DatagramPacket recv = new DatagramPacket(buf,buf.length);
      s.receive(recv);
      System.out.println(
              new String(recv.getData(),0,0,recv.getLength()));

      // Abandonar el grupo multicast
      s.leaveGroup(group);
      }

}
```

1.5. Desarrollo de servidores TCP

- Codificación de mensajes
- Lectura/Escritura de streams de bytes
- Servidores TCP concurrentes
- Opciones de sockets TCP

Codificación de mensajes

- Protocolos estándares sobre TCP
  - Codificación binaria o textual del mensaje
  - Delimitador de final de mensaje (“\r\n”)

```
public class MsgDelim {
  public MsgDelim(BufferReader in) {
    System.setProperty("line.separator","\r\n");
  …}

  void addDelim (String msg, PrintWriter out) throws IOException {
    out.println(msg);
  }

  String readMsgDelim() throws IOException;
     in.readLine();
}
```

Servidores TCP concurrentes

- La mayoría de servidores TCP estándar son concurrentes (implementaciones en C en su mayoría)
- Varias implementaciones
  - Creación de una hebra por cliente
  - Grupo de hebras
  - Uso de la interfaz *Executor*
- Objetivo de diseño
  - Desacoplar la implementación del protocolo de la estrategia de concurrencia

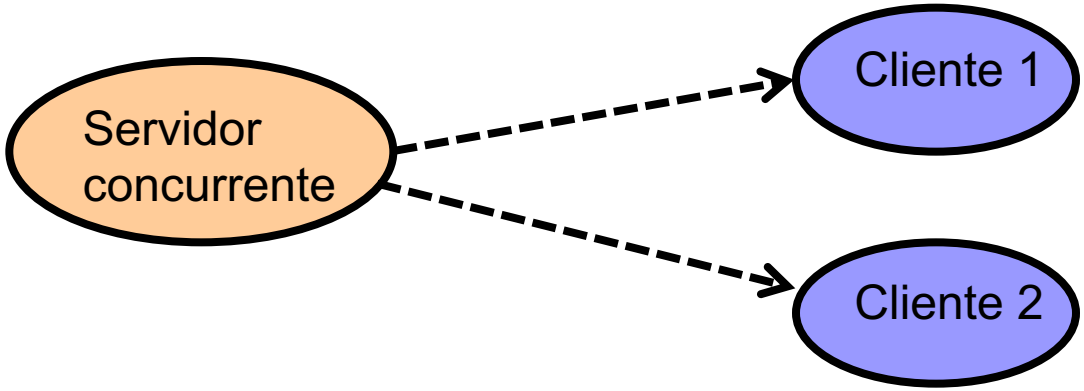
Implementación del protocolo

- Clase que implementa la interfaz *Runnable*
- Implementa la funcionalidad completa del protocolo en un método estático (ej: *handleEchoclient*())
  - Este método debe tener como argumento el socket de transferencia de datos
  - Se ejecuta desde el método run() o se invoca directamente al método estático

```
import java.io.*;
import java.net.*;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EchoProtocol implements Runnable {
  private static final int BUFSIZE = 32; // Tamaño buffer de E/S
  private Socket clntSock;               // Socket de datos
  private Logger logger;                 // Logger del servidor

  public EchoProtocol(Socket clntSock, Logger logger) {
    this.clntSock = clntSock;
    this.logger = logger;
  }

  public static void handleEchoClient(Socket clntSock, Logger logger){
   // Implementacion del protocolo de nivel de aplicacion
  }

  public void run() {
    handleEchoClient(clntSock, logger);
  }
}
```

Una hebra por cliente

- Se crea una hebra a medida que llegan las peticiones de los clientes

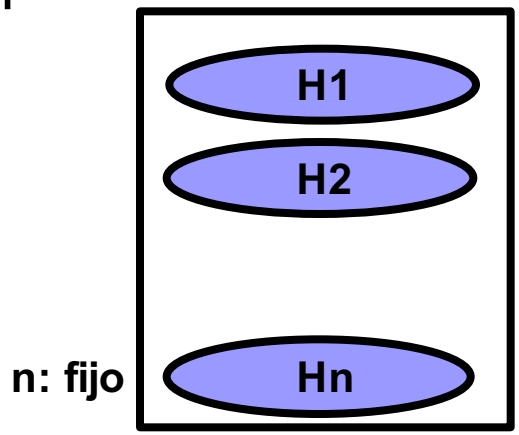


```
import java.io.IOException;
import java.net.*;
import java.util.logging.Logger;

public class TCPEchoServerThread {

  public static void main(String[] args) throws IOException {

    ServerSocket servSock = new ServerSocket(4567);
    // Instanciar un logger para el servidor
    Logger logger = Logger.getLogger("servidorDST-C1");

    // Bucle principal
    while (true) {
      Socket clntSock = servSock.accept(); // Espera una peticion
      // Crea una hebra para gestiona una nueva conexion
      Thread thread = new Thread(new EchoProtocol(clntSock, logger));
      thread.start();
      logger.info("Ejecucion de una nueva hebra " + thread.getName());
    }
  }
}
```

Clase que implementa el protocolo de Eco

Grupo de hebras

- Objetivo
  - Reducir la sobrecarga del sistema por la gestión de muchas hebras (consume muchos recursos y energía)
- Solución
  - Crear un grupo de hebras de tamaño fijo
  - Asignar la petición de un cliente a una hebra existente



```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class TCPEchoServerPool {

  public static void main(String[] args) throws IOException {

    // El numero maximo de hebras se pasa como parametro
    if (args.length != 1) { // Comprobar #parametros
      throw new IllegalArgumentException("Parameter: <Threads>");
    }

    int threadPoolSize = Integer.parseInt(args[1]);

    // Crear el socket pasivo
    final ServerSocket servSock = new ServerSocket(5678);

    // Instanciar un logger para el servidor
    final Logger logger = Logger.getLogger("servidorDST-C2");
```

```
    // Crear un número fijo de hebras
    for (int i = 0; i < threadPoolSize; i++) {
      Thread thread = new Thread() {
        // Clase anonima que espera y sirve una peticion de servicio
        public void run() {
          while (true) {
            try {
              Socket clntSock = servSock.accept();
              EchoProtocol.handleEchoClient(clntSock, logger);
            } catch (IOException ex) {
              logger.log(Level.WARNING, "Client accept failed", ex);
            }
          }
        }
      };
      thread.start();
      logger.info("Created and started Thread = " +thread.getName());
    }
  }
}
```

Método que implementa el protocolo de Eco dentro de la clase EchoProtocol

Uso interfaz *Executor*

- Implementar la gestión del grupo de hebras automáticamente
  - Si una hebra cae, se crea otra igual que la sustituya
  - Si está inactiva durante 60 seg. se quita
  - Etc.
- Es más eficiente que la implementación anterior
  - Consume menos recursos
  - Consume menos energía

```
import java.io.IOException;
import java.net.*;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
import java.util.logging.Logger;

public class TCPEchoServerExecutor {

  public static void main(String[] args) throws IOException {

    // Crea un socket pasivo
    ServerSocket servSock = new ServerSocket(7890);

    Logger logger = Logger.getLogger("servidorDST-C3");

    Executor service = Executors.newCachedThreadPool();

    // Bucle principal
    while (true) {
      Socket clntSock = servSock.accept(); // Espera una peticion
      // Ejecuta una hebra y le asigna el servicio de eco
      service.execute(new EchoProtocol(clntSock, logger));
    }
}
}
```

Clase que implementa el protocolo de Eco

Opciones de sockets

- get/setSoTimeout(int) => SO_TIMEOUT
  - Para la clase *Socket* se puede usar para limitar el tiempo del servicio

```
int timelimit = 10000;
long endTime = System.currentTimeMillis() + timelimit;
int timeBoundMillis = timelimit;
try {
 s.setSoTimeout(timeBoundMillis);

 while ((timeBoundMillis > 0) &&
             ((recvMsgSize = in.read(Buffer)) != -1)) {
  // IMPLEMENTACION DEL SERVICIO
  timeBoundMillis = (int)(endTime - System.currentTimeMillis());
  s.setSoTimeout(timeBoundMillis);
 }
} catch (SocketTimeoutException e) {
   // Finaliza el temporizador de lectura
}
```

  - También puede aplicarse a un *ServerSocket* para limitar el tiempo de espera de una nueva conexión

Opciones de sockets

- get/setKeepAlive() permite chequear si el otro extremo está activo
  - Se envía un mensaje de prueba y si tras N reintentos no se recibe contestación se cierra la conexión
- get/setReceiveBufferSize() – get/setSendBufferSize()
  - Cambia el tamaño del buffer de lectura/escritura
  - Un objeto *Socket* tiene buffers de envío y recepción
  - Un objeto *ServerSocket* tiene sólo buffer de recepción
- Get/setReuseAddress() => SO_REUSEADDR
  - En *ServerSocket* permite reutilizar la dirección (puerto) para evitar esperar al cierre completo del socket pasivo después de ^C

```
ServerSocket servSock = new ServerSocket();
servSock.setReuseAddress(true);
servSock.bind(servPort);  // No deberia fallar
```

Opciones de sockets

- get/setSoLinger() => SO_LINGER
  - Para la clase *Socket*
  - Permite esperar a las confirmaciones antes de cerrar el socket
- set/getTcpNoDelay()
  - En Socket
  - Deshabilita el algoritmo de Nagle en TCP
- get/setOOBInline()
  - Habilita en envío de datos fuera de línea (urgentes)
  - En *Socket*
  - Por defecto no está habilitada

Cliente

```
Socket s=new Socket(IpAddr,6667);
s.sendUrgentData(99);
```

Servidor

```
Socket cs = ss.accept();
cs.setOOBInLine(true);
InputStream in = cs.getInputStream();
in.read(buff);
```

1.6. Patrones de diseño para aplicaciones distribuidas

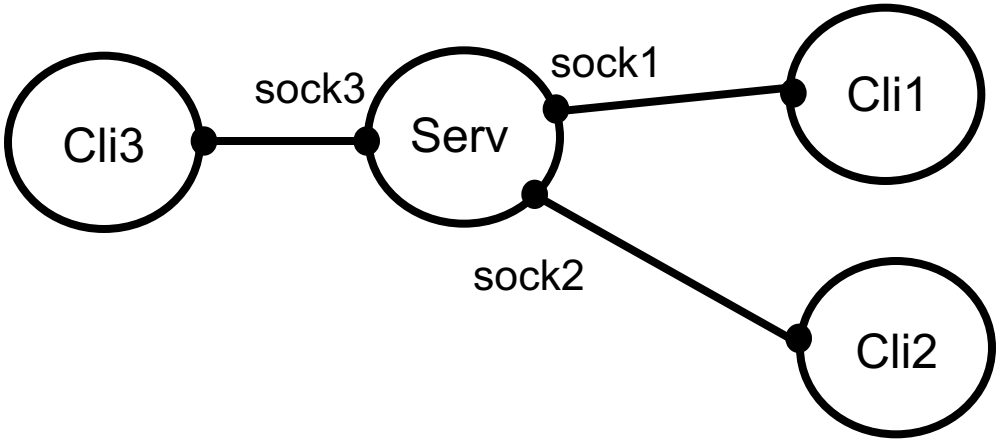
- Patrón multi-cliente
  - Paquete NIO (*Selector*)
- Patrón multi-servidor
  - Paquete NIO (*Selector*)
- Patrón state
- Patrón servidor proxy
- Patrón acceptor

New I/O (NIO): motivación

- Problema de escalabilidad de los servidores multi-hebra
- Imposibilidad de asegurar un orden de ejecución de las hebras
- Servidores multi-cliente
  - Un servidor que atiende a N clientes que participan en el mismo servicio (ej: servicio de chat)
    - Los clientes pueden necesitar actualizar el estado del servidor de forma sincrónica y concurrente.
- Multi-servidores
  - Proporcionan diferentes servicios de forma concurrente utilizando incluso distintos protocolos de comunicación
    - Centralizan el acceso concurrente a todos los servicios de una máquina (ej: el clásico inetd o xinetd más recientemente)

Multiplexión de E/S

- Motivación
  - Lectura de datos procedentes de más de un canal



- Soluciones
  - E/S no bloqueante
  - Uso del selector()

E/S no bloqueante en NIO

- E/S con setSoTimeout()
  - Si un canal no tiene datos hay que esperar que “salte” el temporizador para leer el siguiente canal
    - Ineficiente e implementación compleja con bloques try/catch
- Uso de canales java.nio.channels
  - Tipos de canales NIO (todos son de tipo `Channel`)
    - `DatagramChannel` (para sockets de tipo datagrama)
    - `ServerSocketChannel` (para sockets pasivos)
    - `SocketChannel` (para stream sockets)
  - Lectura no bloqueante
    - Todas las clases `Channel` implementan el método `configureBlocking()`

```
// Crear un SocketChannel y configurarlo como no bloqueante
SocketChannel clntChan = SocketChannel.open();
clntChan.configureBlocking(false);
```

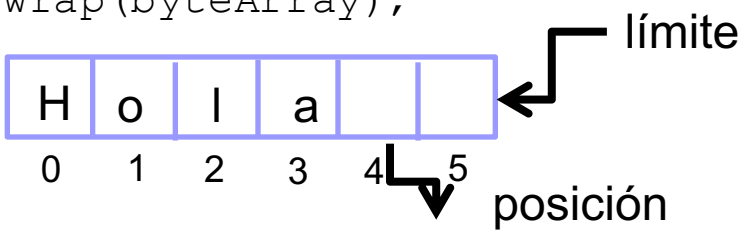
E/S de Channel

- Los Channel no usan streams sino que envían/reciben de buffers
  - Tienen capacidad limitada
  - Es posible conocer cuantos datos se han recibido o enviado
- Nosotros usaremos `ByteBuffer`
  - Creación

```
ByteBuffer buffer = ByteBuffer.allocate(CAPACITY); ó
ByteBuffer buffer = ByteBuffer.wrap(byteArray);
```

  - Lectura/escritura

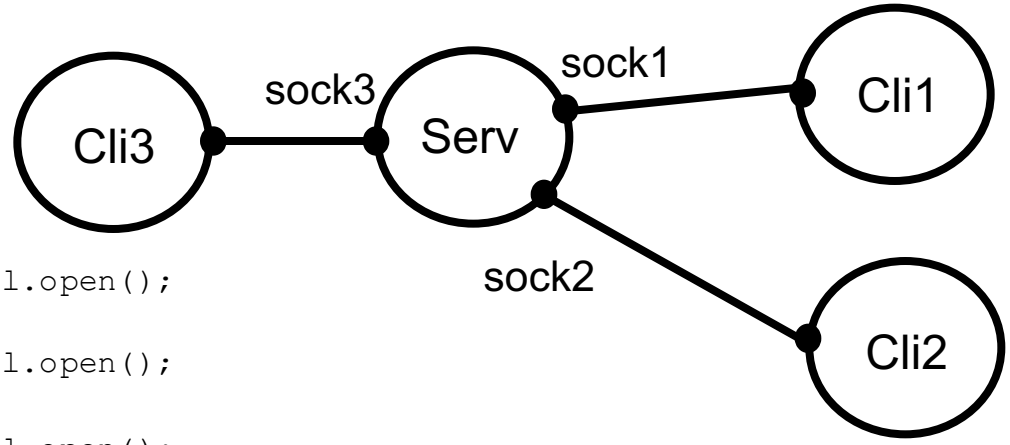
```
buffer.get()
buffer.put()
```



  - Preparar buffer

```
buffer.clear() // poner antes de leer (posicion=0;limite=capacidad)
buffer.flip()  // limite=posicion actual; posicion= 0
buffer.compact() // hace sitio en el buffer
Buffer.rewind()  // Prepara el buffer para re-lectura; posicion=0 y
                 // limite=no cambia
```

Multiplexación de E/S




```
DatagramChannel clntChan1 = DatagramChannel.open();
clntChan1.configureBlocking(false);
DatagramChannel clntChan2 = DatagramChannel.open();
clntChan2.configureBlocking(false);
DatagramChannel clntChan3 = DatagramChannel.open();
clntChan2.configureBlocking(false);

/* lectura por sondeo secuencial de canales */
while(true) {
  …
  if (clntChan1.read(buffer) != 0) {
    /* proceso sock1 */ }
  if (clntChan2.read(buffer) != 0) {
    /* proceso sock2 */ }
  if (clntChan3.read(buffer) != 0) {
    /* proceso sock3 */ }
…
}
```

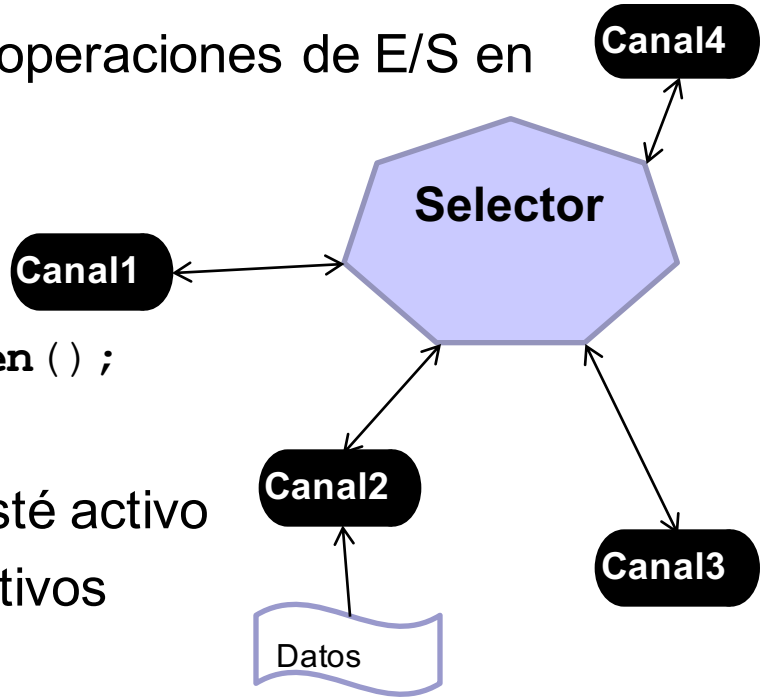
- Problemas lectura no bloqueante de N canales
  - Es muy determinista (el orden de lectura es fijo)
  - El número de descriptores también es fijo
  - Código muy largo y lleno de if

Clase Selector

- Características
  - Evitar esperas innecesarias en servidores multi-cliente y servidores
  - Es un “multiplexor” capaz de atender operaciones de E/S en varios canales a la vez
- Implementación
  - Creación selector

```
Selector selector=Selector.open();
```

  - Registro de canales
  - Esperar que algún canal registrado esté activo
  - Iterar sobre el conjunto de canales activos



Clase Selector

- Registro de canales
  - Registro básico (tipo `Channel`)

```
SelectionKey sk=Achannel.register(selector,
                                  SelectionKey.OP_ACCEPT);
```

Que nos interesa hacer con ese canal

  - Registro con buffer de lectura (tipo `Channel`)

```
Bchannel.register(selector,
                  SelectionKey.OP_READ,ByteBuffer.allocate(1024));
```

Ligar un objeto cualquiera

    - Campos y métodos SelectionKey

```
OP.ACCEPT    isAcceptable()   //Para canales pasivos
OP_CONNECT   isConnectable()  //Para canales activos
OP_READ      isReadable()     //Para canales "legibles"
OP_WRITE     isWritable()     //Para canales "escribibles"
isValid()    //Chequea si una clave key es válida
selector()   //Devuelve el selector donde está la clave key
```

Clase Selector

- Esperar que algún canal registrado en un `Selector` esté activo

```
// Bloquea hasta que al menos un canal este activo
select()
// Bloquea hasta que se cumple el timeout
select(long timeout)
// No bloquea
selectNow()
```

- Iterar sobre el conjunto de canales activos

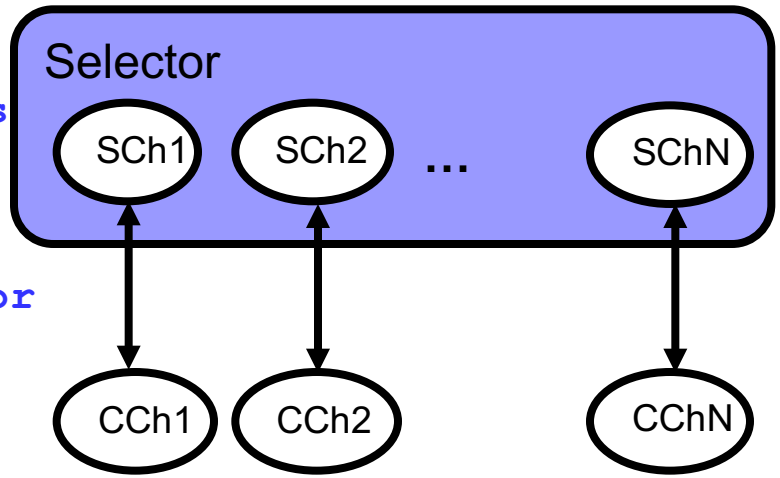
```
// Conjunto de SelectionKey
Set keys = selector.selectedKeys();
```

- Iterar sobre el conjunto de canales del selector

```
// Conjunto de SelectionKey
Set keys = selector.keys();
```

Patrón multi-cliente

```
Abrir selector
Abrir canal pasivo
Ligar el canal al puerto del servidor
Establecer lectura no bloqueante
Registro canal pasivo en selector
for (;;) {
  Llamada select bloqueante
  for (todos los canales activos) {
    quitar canal del conjunto de activos
    if (canal pasivo)
      accept
      agrega canal de lectura al selector
    else
      leo del canal de lectura
      if (fin entrada)
        cierro canal lectura
      else
        proceso datos de entrada según protocolo
  }
}
```




```
public class TCPServerSelector {
  public static void main(String[] args) throws IOException {

    // Abrir el selector
    Selector selector = Selector.open();

    // Abrir el canal pasivo en el selector
    ServerSocketChannel server = ServerSocketChannel.open();
    server.socket().bind(new java.net.InetSocketAddress(9000));
    // Establecer lectura no bloqueante
    server.configureBlocking(false);
    SelectionKey serverkey = server.register(selector,
                        SelectionKey.OP_ACCEPT);
    for (;;) {
      // Llamada bloqueante
      selector.select();
      // Iterar sobre el conjunto de canales activos
      Set keys = selector.selectedKeys();
      for (Iterator i = keys.iterator(); i.hasNext();) {
        SelectionKey key = (SelectionKey) i.next();
        i.remove();
        // SIGUIENTE PAGINA
      }
    }
  }
}
```

```
if (key == serverkey) {
  // Ha llegado una peticion de conexion
  if (key.isAcceptable()) {
    SocketChannel client = server.accept();
    client.configureBlocking(false);
    // Se registra un nuevo canal de lectura
    SelectionKey clientkey = client.register(selector,
              SelectionKey.OP_READ, ByteBuffer.allocate(1024));
  // Ha llegado una petición de lectura
 }
 else {
   if (!key.isReadable())
    continue;
   SocketChannel client = (SocketChannel) key.channel();
   ByteBuffer buffer = (ByteBuffer) key.attachment();
   buffer.clear();
  int bytesread = client.read(buffer);
   if (bytesread == -1) {// Se ha cerrado la conexión?
    key.cancel();
    client.close();
    continue;
  }
  // Procesar mensaje de entrada según protocolo
  // Si no acabe de leer interesa tanto lectura como escritura
 key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
}
```

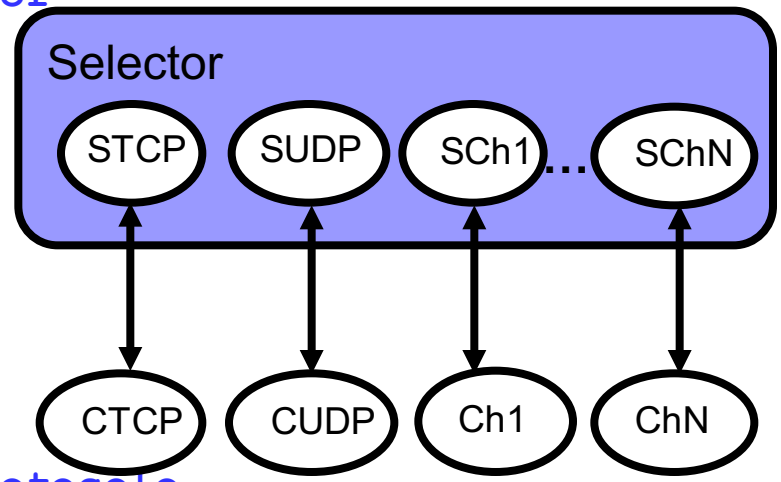
```
// Operación de escritura
if (key.isValid() && key.isWritable()) {
  // Recuperar datos para su escritura
  ByteBuffer buf = (ByteBuffer) key.attachment();
  // Posicionar puntero buffer para escritura
  buf.flip();
  // Obtengo el canal de escritura
  SocketChannel client = (SocketChannel) key.channel();
  client.write(buf);
  if (!buf.hasRemaining()) { // Fin operación escritura
    // Solamente me interesa leer
    key.interestOps(SelectionKey.OP_READ);
  }
buf.compact(); // Hacer sitio para leer mas datos
}
```

```
public class UDPEchoServerSelector {
  public static void main(String[] args) throws IOException {

    ByteBuffer buffer = ByteBuffer.allocate(512);

    // Abrir el selector
    Selector selector = Selector.open();

    // Abrir el canal de tipo datagrama en el selector
    DatagramChannel channel = DatagramChannel.open();
    channel.socket().bind(new InetSocketAddress(6543));
    channel.configureBlocking(false);
    SelectionKey channelkey = channel.register(selector,
                    SelectionKey.OP_READ, new ClientRecord());
    for (;;) {
      // Llamada bloqueante
      selector.select();
      Set keys = selector.selectedKeys();
      for (Iterator i = keys.iterator(); i.hasNext();) {
        SelectionKey key = (SelectionKey) i.next();
        i.remove();
        // SIGUIENTE PAGINA
      }
    }
  }
}
```

```
if (key.isValid()) && (key == channelkey)) {
  // Ha llegado un mensaje
  if (key.isReadable())
    // Procesar mensaje de entrada según protocolo
    DatagramChannel channel = (DatagramChannel) key.channel();
    ClientRecord cr = (ClientRecord) key.attachment();
    cr.buffer.clear();     // Prepara el buffer para lectura
    cr.clientAddress = channel.receive(cr.buffer);
    if (cr.clientAddress != null) {
      // Si acabe de leer entonces paso a escribir
      key.interestOps(SelectionKey.OP_WRITE);
    }
  // Hay que enviar un mensaje
  if (key.isWritable()) {
    // Procesar la escritura de un mensaje
     DatagramChannel channel = (DatagramChannel) key.channel();
     ClientRecord cr = (ClientRecord) key.attachment();
    cr.buffer.flip(); // Prepara el buffer para escritura
    int bs = channel.send(cr.buffer,cr.clientAddress);
    if (bs != 0) { // fin escritura?
       // Si acabe de escribir paso a leer
       key.interestOps(SelectionKey.OP_READ);
    }
  }
```

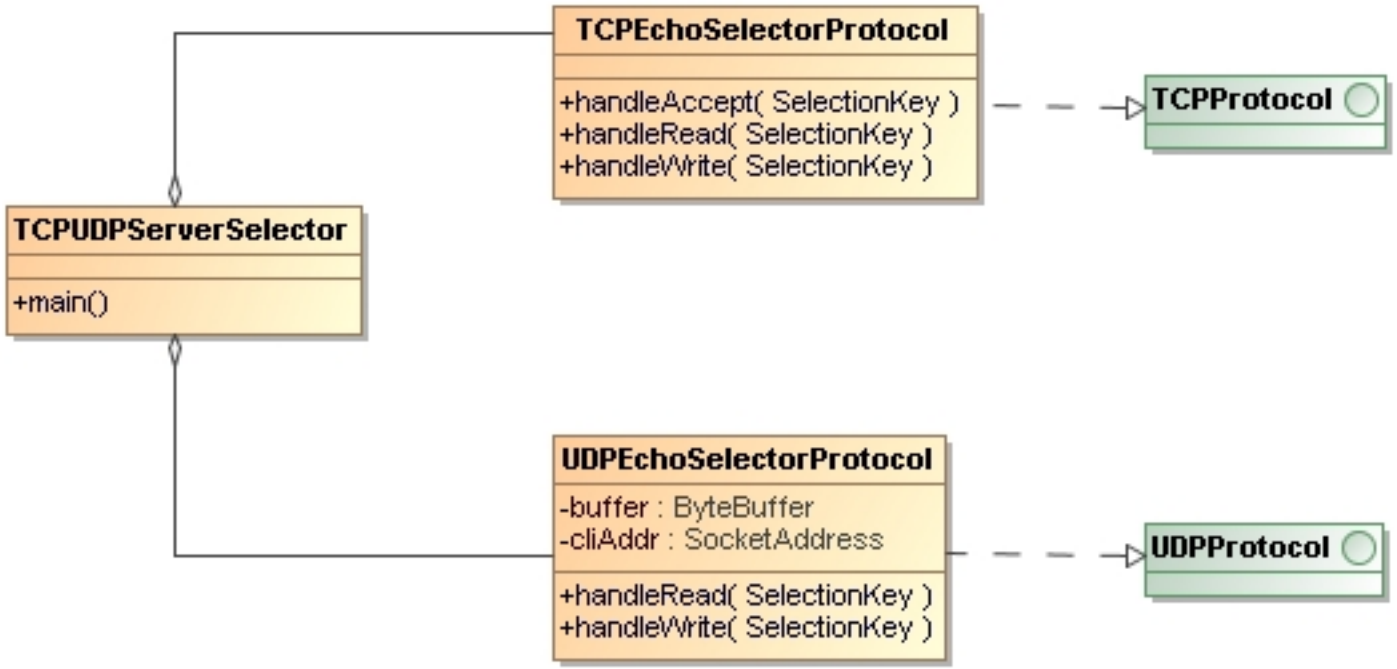
Patrón multi-servidor

```
Abrir canal pasivo TCP
Ligar el canal pasivo al puerto del servidor
Abrir canal UDP y ligarlo al puerto del servidor (DatagramChannel)
Establecer lectura no bloqueante a ambos canales
Abrir selector
Registro canales pasivo TCP y UDP en selector
for (;;) {
  Llamada select bloqueante
  for (todos los canales activos) {
    obtener el canal del conjunto de activos
    if (canal pasivo TCP) {
      accept y establecer lectura no bloqueante
      agrega canal de lectura al selector
    } else if (canal UDP lectura){
        lectura del datagrama
        Procesamiento según protocolo
    } else if (canal TCP lectura) {
      leo del canal de lectura
      if (fin entrada)
        cierro canal lectura
      Proceso datos de entrada según protocolo
    }}}
```



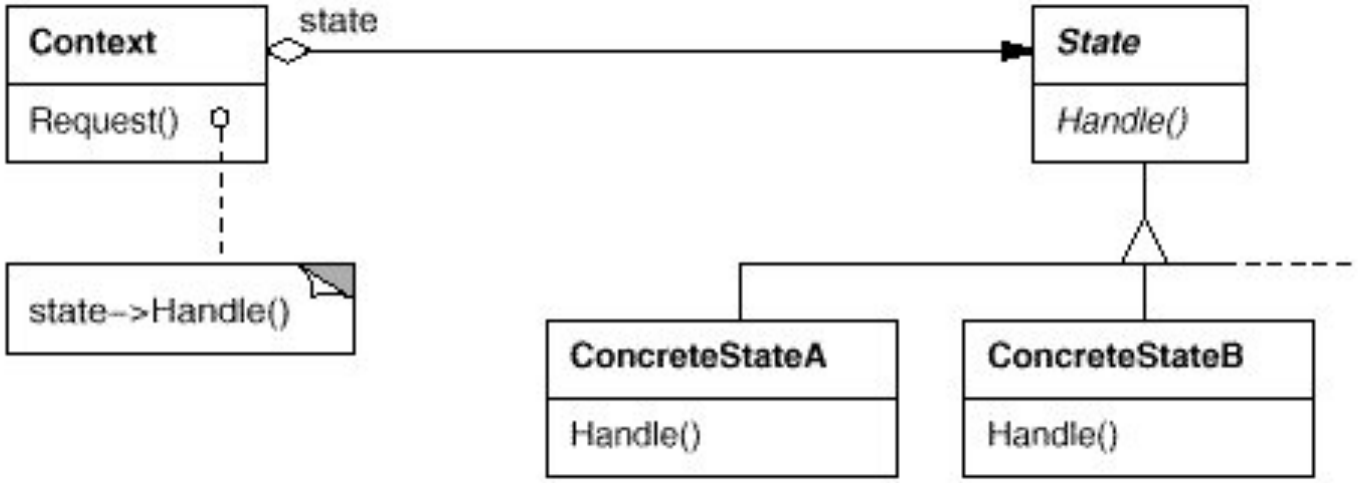
Patrón multiservidor

- Ejemplo protocolos de Echo en TCP y UDP



El patrón State

- Permite invocar el método correcto según el estado y el mensaje de entrada
- Permite implementar diagramas de transición de estados, típicos de la descripción de protocolos



El patrón servidor proxy

- Actúa como apoderado de otro servidor real
  - Seguridad, rendimiento, registro de tráfico, caché, etc.
- Recibe peticiones de servicio y las retransmite al servidor
- Crea una hebra para comunicarse con los clientes y otra para comunicarse con el servidor

```
cs=sockpass.accept(); // Acepta peticion cliente
InputStream in_client=cs.getInputStream(); // datos entrada cliente
OutputStream out_client=cs.getOutputStream(); // datos salida cliente

ss=new Socket(host, port); // conexión con el servidor
InputStream in_server=ss.getInputStream(); // datos entrada servidor
OutputStream out_server=ss.getOutputStream(); // datos salida servidor

// Hebra que gestiona lectura de datos cliente y retransmite al servidor
Thread t=new Thread() {
  public void run() {
    while (! Fin lectura(in_client.read(request))
      out_server.write(request, …);
  }
};
t.start();

// Hebra principal que gestiona peticiones del servidor
while (! Fin lectura (in_server.read(reply))
  out_client.write(reply, …);
```

Implementación parcial de un servidor proxy

El patrón Acceptor

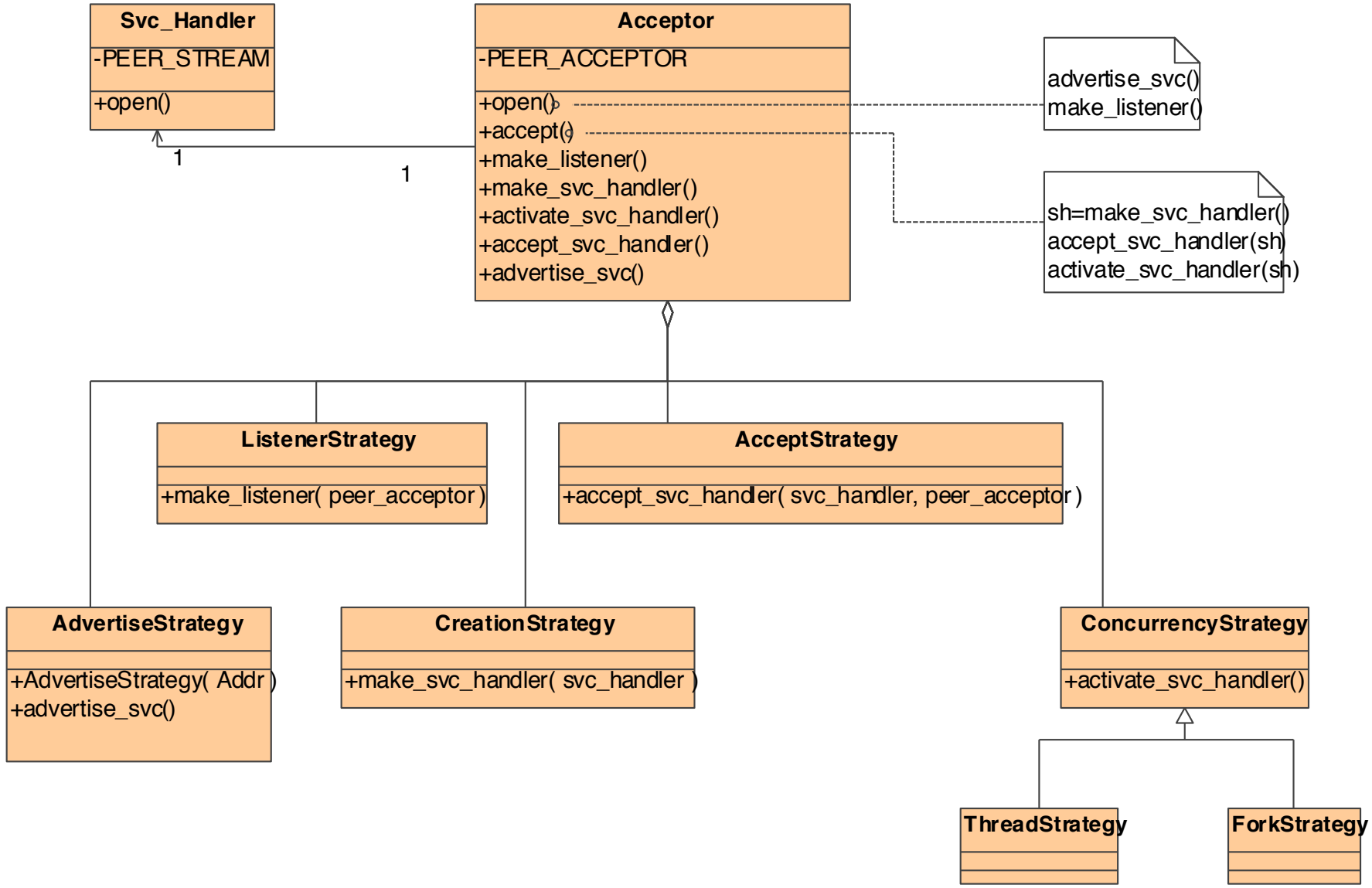
- Permite descoplar la inicialización pasiva de un servicio de las tareas realizadas una vez el servicio es inicializado (proceso servidor)
- Métodos de inicialización
  - Publicar un servicio
  - Estrategia de lectura bloqueante o no bloqueante
  - Creación del manejador de servicio
  - Espera de una conexión o servicio
  - Estrategia de ejecución de un servicio de forma concurrente

El patrón Acceptor

- La ventaja a la hora de aplicar este patrón viene dada por el número de alternativas disponible en cada método de inicialización
- Publicar un servicio
  - Servicio de directorio tipo X.500 (LDAP)
  - Cualquier estrategia para almacenar en el fichero /etc/services el nuevo servicio
- Estrategia de lectura bloqueante/no bloqueante
  - Llamadas bloqueantes (caso normal)
  - Configurar lectura no bloqueante (ej: inicialización de la opción no bloqueante)
  - Configurar lectura mediante selector (lectura de varios sockets)
- Creación del manejador de servicio
  - Crear una instancia por cada nueva conexión (TCP)
  - Reutilizar la misma instancia, con el patrón singleton (UDP)
- Espera de una conexión o servicio
  - Si es orientado a la conexión se haría un accept
  - Si es no orientado a la conexión se haría la lectura de inicio del servicio
- Estrategia de ejecución de un servicio de forma concurrente
  - Creación de procesos hijos con fork() y exec() en C/C++
  - Uso de hebras de ejecución

El patrón Acceptor

- Svc_Handler
  - Esta clase define una interfaz genérica para manejar un servicio
  - Cada instancia encapsula un stream (socket activo a través del cual se comunicará el cliente)
- Acceptor
  - Encargado de inicializar el socket almacenado en la clase Svc_Handler


Svc_Handler
-PEER_STREAM
+open()
Acceptor
-PEER_ACCEPTOR
+open()
+accept()
+make_listener()
+make_svc_handler()
+activate_svc_handler()
+accept_svc_handler()
+advertise_svc()
1
1
advertise_svc()
make_listener()
sh=make_svc_handler()
accept_svc_handler(sh)
activate_svc_handler(sh)
ListenerStrategy
+make_listener(peer_acceptor)
AcceptStrategy
+accept_svc_handler(svc_handler, peer_acceptor)
AdvertiseStrategy
+AdvertiseStrategy(Addr)
+advertise_svc()
CreationStrategy
+make_svc_handler(svc_handler)
ConcurrencyStrategy
+activate_svc_handler()
ThreadStrategy
ForkStrategy

Ventajas patrón acceptor

- Separación de cinco “propiedades” (concerns) en clases independientes
- Evolución independiente de cada propiedad
- Permite crear servidores tanto orientados como no orientados a la conexión
- Mejora la reutilización de la implementación de cada una de las “propiedades” que separa