

Unidad 1

**Sockets de flujo**

Servicios definidos en la capa de  
transporte

# Arquitectura TCP/IP (RFC 1180)



## Aplicación

- HTTP, FTP, TFTP, etc.



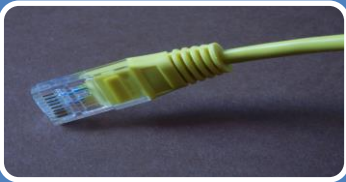
## Transporte

- TCP
- UDP



## Internet

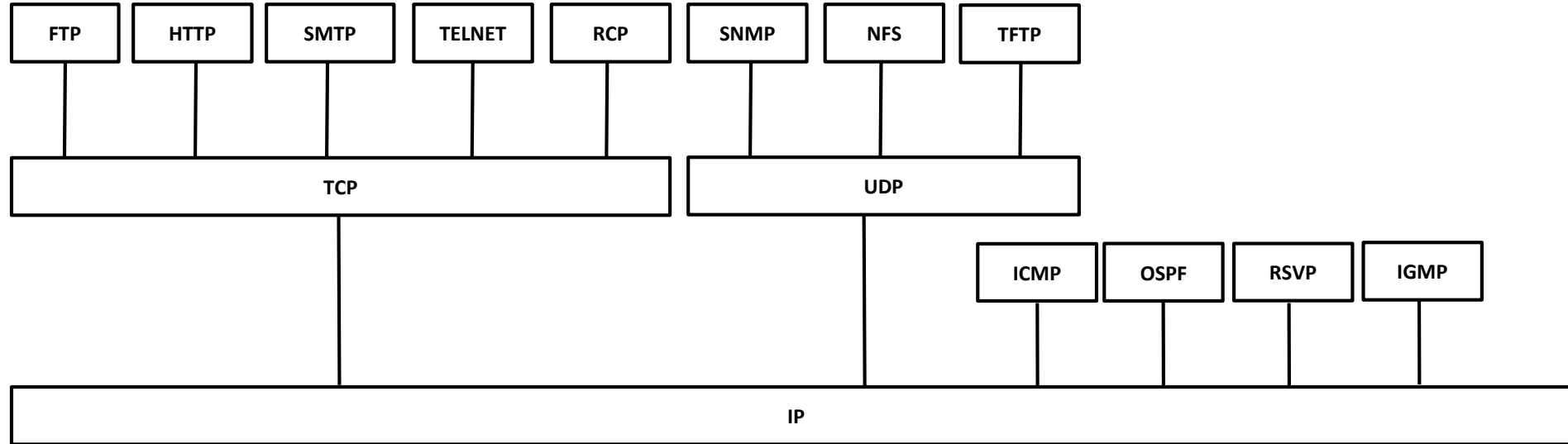
- IP, ICMP, IGMP, ARP, etc.



## Acceso a la red

- LLC
- MAC

# TCP y UDP (RFC 1180)



# UDP (Datagramas)

- *UDP* (RFC 768) es un protocolo que ofrece servicio de transporte de datagramas no orientado a conexión.
- Proporciona un modo de pasar la parte del mensaje de UDP al protocolo de la capa de aplicación (multiplexación).

# Características de UDP (1/2)

- No orientados a conexión
  - Los mensajes de UDP se envían sin la negociación del establecimiento de conexión de TCP (handshake)
- No fiable
  - Los mensajes de UDP se envían como datagramas sin secuencia y sin reconocimiento.
  - El protocolo de aplicación que utiliza los servicios de UDP debe recuperarse de la pérdida de mensajes.
  - Los protocolos típicos de nivel de aplicación que utilizan los servicios de UDP, proporcionan sus propios servicios de fiabilidad o retransmiten periódicamente los mensajes de UDP o tras un periodo de tiempo preestablecido.

# Características de UDP (2/2)

- Proporciona identificación de los protocolos de nivel de aplicación
  - UDP proporciona un mecanismo para enviar mensajes a un protocolo o proceso del nivel de aplicación en un host de una red.
  - El encabezado UDP proporciona identificación tanto del proceso origen como del proceso destino (#puerto)

# Qué no ofrece UDP (1/2)

- Buffer

- UDP no proporciona ningún tipo de buffer de los datos de entrada, ni de salida.
- Es el protocolo de nivel de aplicación quien debe proveer todo el mecanismo de buffer.

- Segmentación

- UDP no proporciona ningún tipo de segmentación de grandes bloques de datos.
- Por lo tanto la aplicación debe enviar los datos en bloques suficientemente pequeños para que los datagramas de IP para los mensajes de UDP, no sean mayores que la MTU de la tecnología de Nivel de Interfaz de Red por la que se envían.
- El tamaño estándar de datos (carga útil) de UDP es de 512 bytes.
- El tamaño máximo de datagrama es de 65536 bytes



# Que no ofrece UDP (2/2)

- Control de flujo

- UDP no proporciona control de flujo ni del extremo emisor, ni del extremo receptor.
- Los emisores de mensajes UDP pueden reaccionar a la recepción de los mensajes de Control de flujo de origen de ICMP, pero no se requiere.

# Usos de UDP (2/2)

- No se requiere fiabilidad por un proceso periódico de anuncios
  - Si el protocolo de Nivel de aplicación publica periódicamente la información, no se requiere un envío fiable.
  - Si se pierde un mensaje, se vuelve a anunciar de nuevo tras el período de publicación.
  - Un ejemplo de protocolo de Nivel de aplicación que usa anuncios periódicos (30 segundos) es el Protocolo de Información de Enrutamiento (RIP – Routing Information Protocol).
- Envío de uno a muchos
  - UDP se utiliza como protocolo de Nivel de transporte siempre que se debe enviar datos de Nivel de aplicación a múltiples destinos mediante direcciones de IP de difusión o multidifusión.
  - TCP se puede usar sólo en envío de uno a uno.
  - Ejemplo: Un envío de señal de video o voz a través de la red de paquetes.

# Encabezado UDP

0	16	31
puerto de origen	puerto de destino	
longitud	checksum	
datos		

# TCP (Flujo)

- El *Protocolo de Control de Transmisión* (TCP – Transmission Control Protocol, RFC 793), es el protocolo de la capa de Transporte que proporciona un servicio de entrega confiable de transferencia de datos de extremo a extremo.
- Y ofrece un método para pasar datos encapsulados mediante TCP a un protocolo de la capa de aplicación

# Características de TCP (1/5)

- Orientado a conexión
  - Antes de poder transferir los datos, dos procesos (local y remoto) deben negociar una conexión TCP mediante un proceso de establecimiento de conexión (handshake).
  - Las conexiones TCP se cierran formalmente mediante el proceso de finalización de conexión TCP.
- Full Duplex
  - Para cada terminal TCP, la conexión TCP está formada por dos canales lógicos: un canal para transmitir datos (salida) y uno para recibir datos (entrada).
  - Con la tecnología adecuada de la capa de Interfaz de Red, la terminal podría transmitir y recibir datos al mismo tiempo.
  - El encabezado TCP contiene el número de secuencia de los datos de salida y un reconocimiento de los datos de entrada.

# Características de TCP (2/5)

- Fiable

- En el transmisor, los datos enviados en una conexión TCP están secuenciados y se espera un reconocimiento afirmativo por parte del receptor.
- Si no se recibe ningún reconocimiento, el segmento se transmite de nuevo.
- En el receptor, los segmentos duplicados se descartan y los segmentos que llegan fuera de secuencia se colocan en la secuencia correcta.
- Siempre se utiliza una suma de comprobación TCP para comprobar la integridad de nivel de bit del segmento TCP.

# Características de TCP (3/5)

- Secuencia de bytes
  - TCP reconoce los datos enviados a través de los canales de entrada y salida como una secuencia continua de bytes.
  - El número de secuencia y el número de reconocimiento en cada encabezado TCP se define en límites de bytes.
  - TCP no reconoce límites de mensajes o registros en la secuencia de bytes.
  - El protocolo de la capa de Aplicación debe proporcionar el análisis correspondiente de la secuencia de bytes de entrada

# Características de TCP (4/5)

- Control de flujo del emisor y del receptor.
  - Para evitar el envío de demasiados datos a la vez y la saturación de la red IP.
  - TCP implementa control de flujo del emisor que, gradualmente, escala la cantidad de datos a la vez.
  - Para evitar que el emisor envíe datos que el receptor no puede almacenar en buffer.
  - TCP implementa control de flujo del receptor que indica la cantidad de espacio libre en el buffer del receptor.

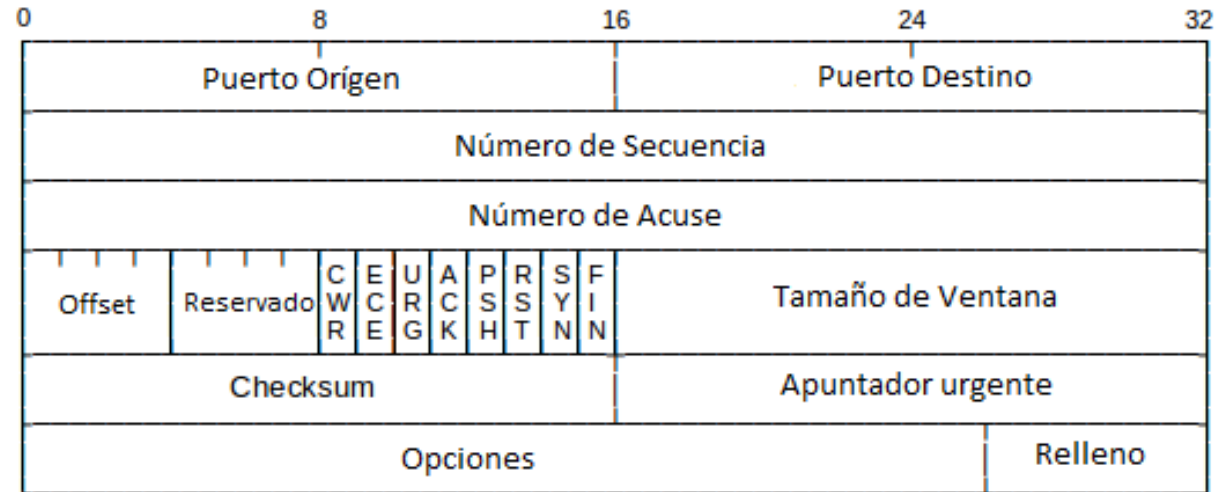


# Características de TCP (5/5)

- Entrega de uno a uno
  - Las conexiones de TCP son un circuito lógico punto a punto entre dos protocolos de la capa de Aplicación.
  - TCP no proporciona un servicio de uno a varios.

Normalmente, TCP se utiliza cuando el protocolo de la capa de aplicación requiere un servicio de transferencia de datos confiable y el protocolo de aplicación no proporciona este tipo de servicio.

# Encabezado TCP



SYN = Sincronización

FIN = Finalización

RST = Reinicio

PSH = Envío

ACK = Acuse

URG = Urgente

ECE = Congestión Explícita Experimentada

(*Explicit-Congestion-Notification-Echo*)

//ACK

CRW = Reducción de Ventana por Congestión

//Emisor

Modelo cliente/servidor

# Modelo cliente-servidor

- Los términos de cliente y servidor se refieren a los roles que realizan
- El cliente inicia la comunicación
- El servidor espera pasivamente y responde a la llamada del cliente.
- Juntos conforman la aplicación

# Modelo cliente/servidor



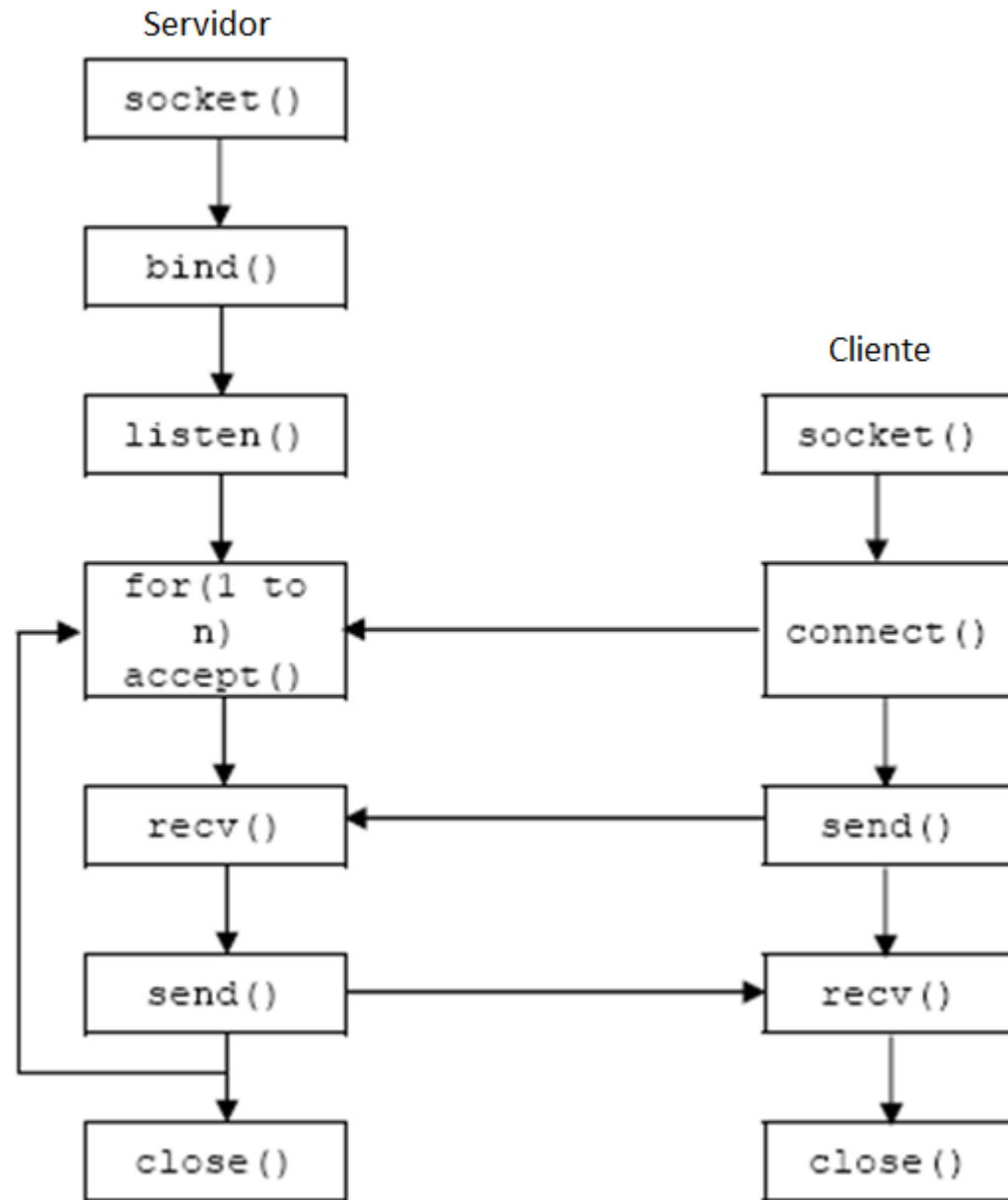
# Funcionamiento del cliente

- Crea un socket
- Establece la conexión con el servidor
- Se comunica enviando y recibiendo mensajes
- Cierra la conexión

# Funcionamiento del servidor

- Crea un socket y lo asocia a un puerto local determinado
- Espera hasta que alguien se conecte
- Repite de forma ininterrumpida:
  - Acepta cada nueva conexión al socket
  - Se comunica enviando y recibiendo mensajes con el cliente
  - Cierra la conexión con cada cliente

# Diagrama de flujo





Conexiones en el dominio de  
internet

¿Qué es un socket?



# Sockets

- Un *socket* es una abstracción
- Representa un extremo en una comunicación bidireccional entre dos aplicaciones que se comunican a través de la red.

# Sockets

- Diferentes tipos de sockets corresponden a diferentes tipos de protocolos.
- Solo trabajaremos con sockets de TCP/IP
- Sockets de flujo representan el extremo de una conexión TCP
- Sockets de datagrama son un servicio de mejor esfuerzo para el envío individual de datos.
- Un socket TCP/IP se identifica con un número de puerto y una dirección IP

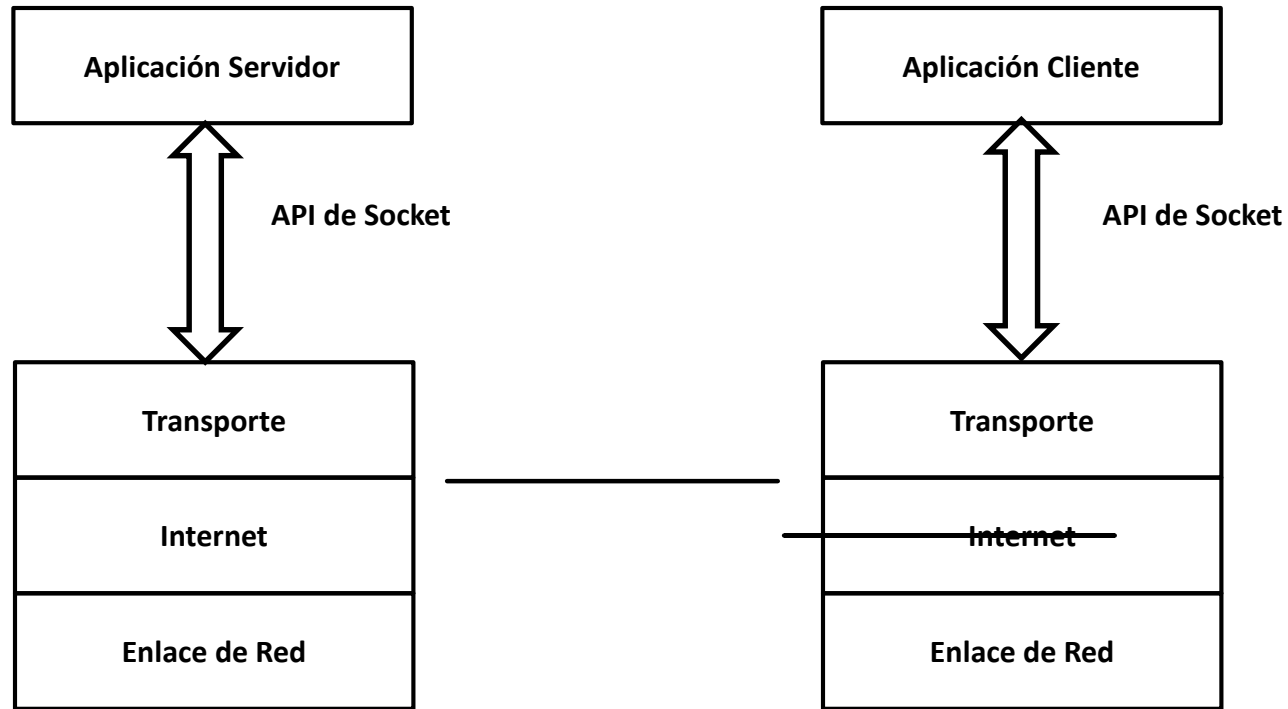
# Sockets bloqueantes y no bloqueantes

- No se trata de sockets diferentes realmente, son solo opciones para las formas en las que trabajan
- Los sockets bloqueantes son aquellos que se quedan esperando hasta que existe información para establecer una conexión, leer o escribir un mensaje
- Los sockets no bloqueantes interrogan si hay datos para procesar y en caso de que no se así, continúan con el código
- El socket no bloqueante se definen modificando sus opciones

# API de sockets

- Interfaz de programación de aplicaciones
- Conjunto de subrutinas, funciones y procedimientos (o métodos) que ofrece cierta biblioteca para ser utilizado por otro software.

# Sockets API



Sockets orientados a conexión  
bloqueantes



# Socket de flujo bloqueantes

- Es el tipo de socket que utiliza el protocolo TCP y por tanto tiene todas las características relacionadas

# API java

*<https://docs.oracle.com/javase/8/docs/api>*

# Clase Socket

- Implementa un socket de flujo del lado del cliente
- Se le llama simplemente socket
- Se encuentra en el paquete **java.net**

# Constructores principales de Socket

- **Socket();** crea un socket de flujo desconectado
- **Socket(InetAddress address, int port);** Crea un socket de flujo y lo conecta a un número de puerto en una IP definida
- **Socket(InetAddress address, int port, InetAddress localAddress, int localPort);** Crea un socket de flujo, ligado a una dirección y puerto local y lo conecta a un número de puerto en una IP definida remota

# Métodos principales de Socket

- `bind(SocketAddress bindport);` vincula al socket con algún puerto local.
- `close();` Cierra el socket.
- `connect(SocketAddress dst);` conecta al socket con el servidor
- `connect(SocketAddress dst, int t);` conecta al socket con el servidor definiendo un tiempo máximo para la conexión

# Clase ServerSocket

- Implementa un socket de servidor de flujo
- Una instancia de esta clase espera por solicitudes de conexión en la red
- Se encuentra en el paquete **java.net**

# Constructores principales de ServerSocket()

- `ServerSocket();` crea un socket de servidor.
- `ServerSocket(int pto);` crea un socket de servidor asociado a un puerto.
- `ServerSocket(int pto, int backlog);` crea un socket de servidor ligado a un puerto con una cola de conexiones específica.
- `ServerSocket(int pto, int backlog, InetAddress dir_local);` crea un socket de servidor ligado a un puerto con una cola de conexiones específica

# Métodos principales de ServerSocket

- **accept()**; acepta una conexión a través de la red.
- **bind(SocketAddress local)**; vincula al **ServerSocket** con una dirección IP y número de puerto específico.
- **close()**; Cierra el socket.



ServerSocket() y Bind()

```
ServerSocket s = new ServerSocket();
```

```
InetSocketAddress dir = new
```

```
InetSocketAddress(1234);
```

```
s.bind(dir);
```

ó

```
ServerSocket s = new ServerSocket(1234);
```

# Flujos en java

# Flujos en java

- Paquete **java.io**



- Flujos orientados a byte / orientados a carácter

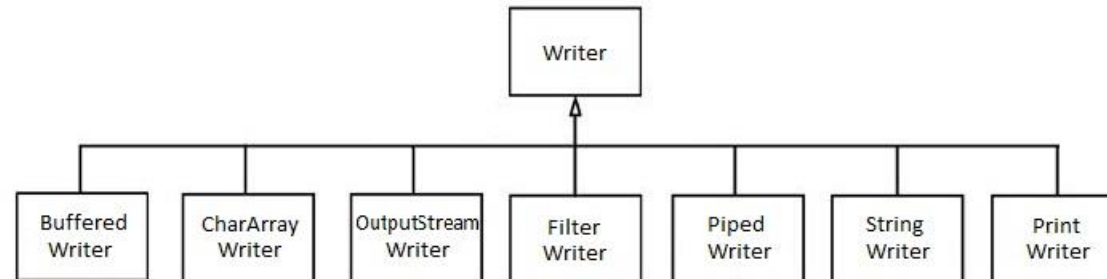
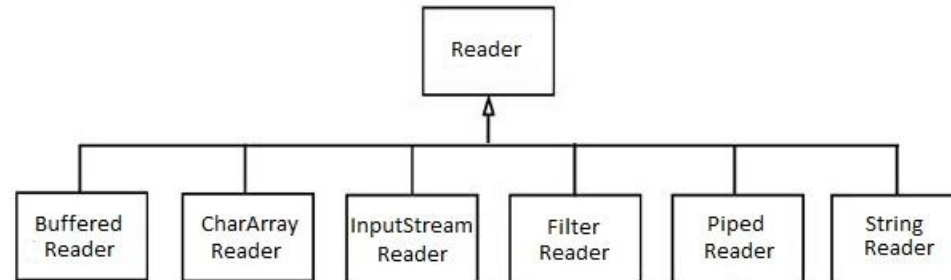
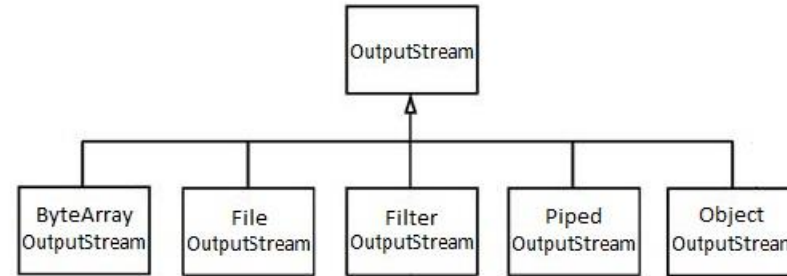
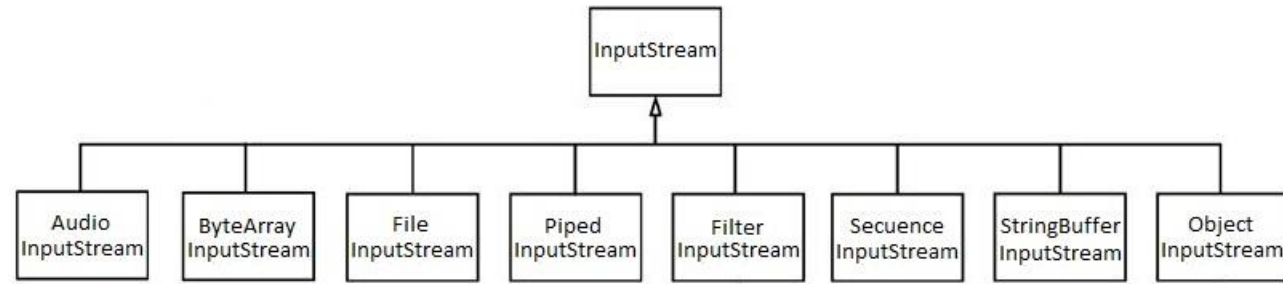
# Flujos orientados a byte

- Byte (8bits)
- Más primitivos y portables
- Los demás flujos lo usan
- Flujo de bajo nivel
- InputStream y OutputStream

# Flujos orientados a carácter

- char (16 bits)
- Codificación unicode
- Ideal para texto plano
- Reader y Writer

# Diagrama de clases principales



# Lectura y escritura

- Abrir
- Leer o escribir
- Cerrar

# Lectura, InputStream

- `int read();` Lee el próximo byte del flujo representado en un entero. Devuelve -1 si no quedan más datos que leer.
- `int read(byte[] b);` Lee un arreglo de bytes del flujo.
- `int read(byte[] b, int off, int tam);` Lee un arreglo de bytes del flujo, desde y hasta la posición indicada



# Lectura, Reader

- **int read()** – Lee el próximo carácter del flujo representado en un entero. Devuelve -1 si no quedan ms datos que leer.
- **int read(char[] cbuf)** – Lee un arreglo de caracteres del flujo.
- **int read(char[] cbuf, int off, int len)** – Lee un arreglo de caracteres del flujo, desde y hasta la posición indicada.

# Escritura, OutputStream

- `void write(int b);` Escribe un solo byte en el flujo.
- `void write(byte[] b);` Escribe un arreglo de bytes en el flujo.
- `void write(byte[] b, int off, int len);` Escribe una porción de un arreglo de bytes en el flujo.

# Escritura, Writer

- `void write(int c);` Escribe un solo carácter en el flujo.
- `void write(char[] cbuf);` Escribe un arreglo de caracteres en el flujo.
- `void write(char[] cbuf, int off, int len);` Escribe una porción de un arreglo de caracteres en el flujo

# Entrada y salida estándar

- Clase **System** dentro de **java.lang**
- **InputStream in (InputStream);** Flujo de entrada estándar. Típicamente corresponde al teclado.
- **PrintStream out (OutputStream);** Flujo de salida estándar. Típicamente corresponde a la pantalla.
- **PrintStream err (OutputStream);** Flujo de salida estándar de errores. Típicamente corresponde a la pantalla.
- Pueden ser redirigidos

# Ejemplo

- Realizar una aplicación con una arquitectura cliente/servidor en java con sockets bloqueantes
- El cliente se conecta con el servidor y recibe un mensaje

# Programa de eco

**Cliente**

**Servidor**

## Ejemplo 2: envío de archivos

- Crear una aplicación para el envío de un archivo desde el cliente al servidor
- Se usará un socket orientado a conexión bloqueante
- Los archivos podrán ser de texto o binarios

# Tarea

- Modificar el archivo anterior para que permita el envío de múltiples archivos



# Sockets en C

Sockets bloquantes

# Bibliotecas más utilizadas

- <sys/types.h>: tipos de datos utilizados(pthread\_attr\_t, size\_t, socklen\_t, etc.)
- <sys/socket.h>: macros: SOCK\_STREAM, SOCK\_DGRAM, SOL\_SOCKET, etc. Prototipos: socket(), bind(), send(), recv, accept, etc.
- <stdlib.h>: prototipos: atoi(), malloc(), exit()
- <stdio.h>: prototipos: fopen(), fdopen(), fflush(), scanf(), printf(), etc.
- <netdb.h>: prototipos: freeaddrinfo(), getaddrinfo(), getnameinfo(), etc.

# Estructura sockaddr\_in //<netinet/in.h>

```
struct sockaddr_in {  
    short sin_family; // AF_INET (IPv4), AF_UNIX, AF_LOCAL, etc.  
    unsigned short sin_port; // ej. htons(2000)  
    struct in_addr sin_addr; // ver estructura in_addr  
    char sin_zero[8]; // poner en cero's  
};
```

```
struct in_addr {  
    unsigned long s_addr; // load with inet_aton()  
};
```

# Estructura addrinfo //<netdb.h>

```
struct addrinfo {
    int ai_flags;          // AI_PASSIVE, AI_CANONNAME, AI_NUMERICAL_HOST, etc.
    int ai_family;         // AF_INET, AF_INET6, AF_UNSPEC, AF_BTH, AF_IRDA, etc.
    int ai_socktype;       // SOCK_STREAM, SOCK_DGRAM, SOCK_RAW, SOCK_RDM
    int ai_protocol;       // 0, IPPROTO_TCP, IPPROTO_UDP
    socklen_t ai_addrlen;  // sizeof(ai_addr)
    struct sockaddr *ai_addr; // struct sockaddr_in/sockaddr_in6
    char *ai_canonname;     // nombre canónico
    struct addrinfo *ai_next; // sig. Nodo de lista ligada
};
```

**Nota:** ai\_flags=PASSIVE && nodo=NULL (en func. getaddrinfo( ) ) para hacer bind( )

# Función getaddrinfo()

//<sys/types.h>, <sys/socket.h>,  
//<netdb.h>

```
int getaddrinfo(const char *nodo, //ej. "www.pc1.net" ó "127.0.0.1"  
               const char *servicio, //ej. "FTP", ó "21"  
               const struct addrinfo *i, // apunta a estructura con info importante  
               struct addrinfo **res); //apuntador a lista ligada con el resultado de la consulta
```

- Valor devuelto

0 = éxito

EAI\_ADDRFAMILY= El host no tiene una dirección IP en la familia de direcciones

EAI\_AGAIN= El nombre de host devolvió una falla temporal (reintentar)

EAI\_BADFLAGS=*i.ai\_flags* contiene una bandera inválida/está habilitada la bandera

AI\_CANNONNAME y el nombre es NULL

EAI\_FAIL= Falla permanente

EAI\_FAMILY= familia de direcciones no soportada

EAI\_NONAME=Nodo o servicio desconocidos, o ambos son NULL, o están puestas las

banderas AI\_NUMERICSERV o AI\_NUMERICHOST y uno/ambos de ellos no es numérico

# Ejemplo 1 //para un servidor

```
int r;
struct addrinfo i, *lista;
memset(&i,0,sizeof(i));
i.ai_family = AF_INET6; // IPv4 ó IPv6
i.ai_socktype = SOCK_STREAM;
i.ai_flags = AI_PASSIVE; //solo para el servidor o cuando se use bind
if((r=getaddrinfo(NULL,"5678", &i,&lista))!=0){
    fprintf(stderr,"error:%s\n",gai_strerror(r));
    exit(1);
}
// se crea el socket y cuando ya no se necesite la lista se elimina
freeaddrinfo(lista);
```

## Ejemplo2 //para un cliente

```
int r;
struct addrinfo i, *lista;
memset(&i,0,sizeof(i));
i.ai_family = AF_UNSPEC; // IPv4 ó IPv6
i.ai_socktype = SOCK_STREAM;
if((r=getaddrinfo("200.1.2.3","5678", &i,&lista))!=0){
    fprintf(stderr,"error:%s\n",gai_strerror(r));
    exit(1);
}
// se crea el socket y cuando ya no se necesite la lista se elimina
freeaddrinfo(lista);
```

# Función socket() //<sys/socket.h>

- int socket(int dominio, int tipo, int protocolo)

```
int sd;
struct addrinfo i, *r, *p;
memset(&i, 0, sizeof(i)); //inicio
i.ai_family = AF_INET6; /* Permite IPv4 or IPv6 */
i.ai_socktype = SOCK_STREAM;
i.ai_flags = AI_PASSIVE; // utilizado para hacer el bind
i.ai_protocol = 0; /* Any protocol */
i.ai_canonname = NULL;
i.ai_addr = NULL;
i.ai_next = NULL;
if ((rv = getaddrinfo(NULL, pto, &i, &r)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
}
//if
for(p = r; p != NULL; p = p->ai_next) {
    if ((sd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }
    //if
    break;
}
//for
```



# Familia de direcciones (1/2)

AF_LOCAL	Es otro nombre para AF_UNIX
AF_INET	Protocolo internet DARPA (TCP/IP)
AF_INET6	Protocolo internet versión 6
AF_PUP	Antigua red Xerox
AF_CHAOS	Red Chaos del MIT
AF_NS	Arquitectura Xerox Network System
AF_ISO	Protocolos OSI
AF_ECMA	Red European Computer Manufactures
AF_DATAKIT	Red Datakit de AT&T
AF_CCITT	Protocolos del CCITT, por ejemplo X.25
AF_SNA	System Network Architecture (SNA) de IBM
AF_DECnet	Red DEC

# Familia de direcciones (2/2)

AF_IMPLINK	Antigua interfaz de enlace 1822 Interface Message Processor
AF_DLI	Interfaz directa de enlace
AF_LAT	Interfaz de terminales de red de área local
AF_HYLINK	Network System, C�rporation Hyperchannel
AF_APPLETALK	Red AppleTalk
AF_ROUTE	Comunicaci�n con la capa de encaminamiento del n�cleo
AF_LINK	Acceso a la capa de enlace
AF_XTP	eXpress Transfer Protocol
AF_COIP	Connection-oriented IP (ST II)
AF_CNT	Computer Network Tecnology
AF_IPX	Protocolo Internet de Novell

# Tipos de semántica de la comunicación

- **SOCK\_STREAM**, sockets de flujo
- **SOCK\_DGRAM**, sockets de datagrama
- **SOCK\_RAW**, sockets crudos
- **SOCK\_SEQPACKET**, conector no orientado a conexión pero fiable de longitud fija (solo en **AF\_NS**)
- **SOCK\_RDM**, conector no orientado a conexión pero fiable y secuencial (no implementado pero se puede simular a nivel de capa de usuario)

# Función bind()

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
int bind(int sd, const struct sockaddr *addr, socklen_t  
addrlen);
```

- Valor devuelto:

0 = éxito  
-1 = error

# Ejemplo bind()

```
int sd;
struct addrinfo i, *r, *p;
memset(&i, 0, sizeof (i)); //inicio
i.ai_family = AF_INET6; /* Permite IPv4 or IPv6 */
i.ai_socktype = SOCK_STREAM;
i.ai_flags = AI_PASSIVE; // utilizado para hacer el bind
i.ai_protocol = 0; /* Any protocol */
i.ai_canonname = NULL;
i.ai_addr = NULL;
i.ai_next = NULL;
if ((rv = getaddrinfo(NULL, pto, &i, &r)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
} //if
for(p = r; p != NULL; p = p->ai_next) {
    if ((sd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    } //if
    if (bind(sd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sd);
        perror("server: bind");
        continue;
    } //if

    break;
} //for
```

# Función listen()

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
int listen(int sd, int backlog);
```

//backlog tiene un máximo definido en

SOMAXCONN=128 en /usr/src/linux/net/ipv4/af\_inet.c.

Valor de retorno: 0 = éxito  
-1 = error

5 = mal desempeño en webserver

(/usr/src/linux/socket.h en kernels 2.x )

```
if(listen(sd,80)==-1) {  
    perror("error en func. Listen()\n");  
    close(sd);  
    exit(1);  
}
```

# Función accept()

`#include <sys/socket.h>`

`int accept (int sd, struct sockaddr *dir,  
socklen_t *tam_dir)`

Valor devuelto:  $\left\{ \begin{array}{l} >0 = \text{éxito} \\ -1 = \text{error} \end{array} \right.$

```
char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];
struct sockaddr_storage cdir;
socklen_t ctam = sizeof(cdir);
cd = accept(sd, (struct sockaddr *)&cdir, &ctam);
if (cd == -1) {
    perror("accept");
    continue;
}
if(getnameinfo((struct sockaddr *)&cdir, sizeof(cdir), hbuf,
sizeof(hbuf), sbuf, sizeof(sbuf), NI_NUMERICHOST | NI_NUMERICSERV) ==
0)
    printf("cliente conectado desde %s:%s\n", hbuf, sbuf);
```

# Función write()

#include <unistd.h>

int write(int sd, const void \*buf, size\_t  
tam)

Valor devuelto {  
>0 = #bytes enviados  
-1 = error  
0 = socket cerrado

```
char *msj = "un mensaje";
int n = write(cd, msj, strlen(msj)+1);
if(n<0)
    perror("Error en la función write\n");
else if(n==0){
    perror("Socket cerrado\n");
    exit(1);
}
int v=2;
n = write(cd, &v, sizeof(v));
```

```
float v2= 5.1f;
Char b[10];
memset(b, 0, sizeof(b));
sprintf(b, "%f", v2);
n=write(cd, b, strlen(b)+1);
```

...

```
struct dato{
    char nombre[30];
    char apellido[25];
    int edad;
};

struct dato *o;
o = (struct dato *)malloc(sizeof (struct dato));

O->nombre="Juan";
O->apellido="Perez";
O->edad=htonl(23);
n = write(cd, (const char*)o, sizeof(struct dato));

...
free(o);
```



# Función send()

#include <sys/socket.h>

int send(int sd, const void \*buf, size\_t  
tam, int bandera)

char \*msj = "un mensaje";

int n = send(cd, msj, strlen(msj)+1);

if(n<0)

    perror("Error en la función send()\n");

else if(n==0){

    perror("Socket cerrado\n");

    exit(1);

}

int v=2;

n = send(cd, &v, sizeof(v));

...

0 = prioridad default  
MSG\_OOB= alta prioridad

Valor devuelto:

>0 = #bytes enviados  
-1 = error  
0 = socket cerrado

# Función read()

#include <unistd.h>

int read(int sd, const void \*buf, size\_t

tam)

```
char buf[100];
int n = read(cd,buf, sizeof(buf));
if(n<0)
    perror("Error en la función read()\n");
else if(n==0){
    perror("Socket cerrado\n");
    exit(1);
}
int v;
n = read(cd,&v,sizeof(v));
...
```

```
struct dato{
    char nombre[30];
    char apellido[25];
    int edad;
};
```

```
Chat b[200];
bzero(b,sizeof(b));
n = read(cd,b,sizeof(b));
struct dato *o = (struct dato *)b;
```

Valor devuelto:

>0 = #bytes leídos  
-1 = error  
0 = socket cerrado

# Función recv()

`#include <sys/socket.h>`

`int recv(int sd, const void *buf, size_t  
tam, int bandera)`

`char buf[100];`

`int n = recv(cd, buf, sizeof(buf), 0);`

`if(n<0)`

`perror("Error en la función recv\n");`

`else if(n==0){`

`perror("Socket cerrado\n");`

`exit(1);`

`}`

`int v;`

`n = recv(cd, &v, sizeof(v), MSG_OOB);`

`...`

0 = prioridad default  
MSG\_OOB= alta prioridad

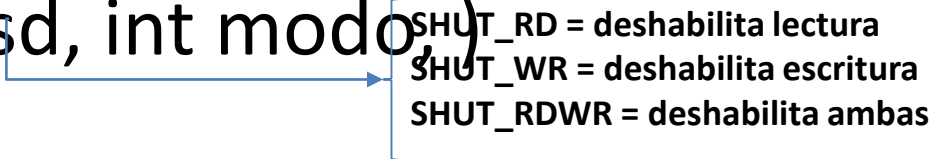
Valor devuelto:

>0 = #bytes leídos  
-1 = error  
0 = socket cerrado

# Función shutdown()

`#include <sys/socket.h>`

`int shutdown(int sd, int modo);`



- SHUT\_RD = deshabilita lectura
- SHUT\_WR = deshabilita escritura
- SHUT\_RDWR = deshabilita ambas

Valor devuelto:  $\begin{cases} 0 = \text{éxito} \\ -1 = \text{error} \end{cases}$

```
cd = accept(sd, (struct sockaddr *)&cdir, &ctam);  
if (shutdown(cd, SHUT_RD) != 0)  
    perror("No fue posible deshabilitar lectura");
```

# Función close()

```
#include <unistd.h>
```

```
int close(int sd)
```

Valor devuelto:  $\begin{cases} 0 = \text{éxito} \\ -1 = \text{error} \end{cases}$

# Función connect()

```
#include <sys/socket.h>
```

```
#include <sys/types.h>
```

```
int connect(int sd, const struct sockaddr *dir, socklen_t  
tam_ref);
```

Valor devuelto:  $\begin{cases} 0 = \text{éxito} \\ -1 = \text{error} \end{cases}$

# Ej. connect()

```
int op = 0;
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((cd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("client: socket");
        continue;
    }

    /*if (setsockopt(cd, IPPROTO_IPV6, IPV6_V6ONLY, (void *)&op, sizeof(op)) == -1) {
        perror("setsockopt no soporta IPv6");
        exit(1);
    }*/

    if (connect(cd, p->ai_addr, p->ai_addrlen) == -1) {
        close(cd);
        perror("client: connect");
        continue;
    }

    break;
} //for
```

# Sockets de datagrama bloqueantes en C



# Función socket() //<sys/socket.h>

- `int socket(int dominio, int tipo, int protocolo)`

```
int sd;
struct addrinfo i, *r, *p;
memset(&i, 0, sizeof(i)); //inicio
i.ai_family = AF_INET6; /* Permite IPv4 or IPv6 */
i.ai_socktype = SOCK_DGRAM;
i.ai_flags = AI_PASSIVE; // utilizado para hacer el bind
i.ai_protocol = 0; /* Any protocol */
i.ai_canonname = NULL;
i.ai_addr = NULL;
i.ai_next = NULL;
if ((rv = getaddrinfo(NULL, pto, &i, &r)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
} //if
for(p = r; p != NULL; p = p->ai_next) {
    if ((sd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    } //if
    break;
} //for
```

# Función bind()

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
int bind(int sd, const struct sockaddr *addr, socklen_t  
addrlen);
```

- Valor devuelto:

0 = éxito  
-1 = error

# Ejemplo bind()

```
int sd;
struct addrinfo i, *r, *p;
memset(&i, 0, sizeof (i)); //inicio
i.ai_family = AF_INET6; /* Permite IPv4 or IPv6 */
i.ai_socktype = SOCK_DGRAM;
i.ai_flags = AI_PASSIVE; // utilizado para hacer el bind
i.ai_protocol = 0; /* Any protocol */
i.ai_canonname = NULL;
i.ai_addr = NULL;
i.ai_next = NULL;
if ((rv = getaddrinfo(NULL, pto, &i, &r)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
} //if
for(p = r; p != NULL; p = p->ai_next) {
    if ((sd = socket(p->ai_family, p->ai_socktype, p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    } //if
    if (bind(sd, p->ai_addr, p->ai_addrlen) == -1) {
        close(sd);
        perror("server: bind");
        continue;
    } //if

    break;
} //for
```

# Función sendto()

`#include <sys/socket.h>`

`ssize_t sendto(int sd, const void *buf, size_t tam, int bandera, const struct sockaddr *dst, socklen_t tam)`

Valor devuelto: {  
    >0 = #bytes enviados  
    -1 = error  
    0 = socket cerrado

└─> {  
    0 = prioridad default  
    MSG\_OOB= alta prioridad

# Ej. sendto()

```
char *msj ="un mensaje";
```

```
int v1=htonl(5);
```

```
float v2 = 3.0f;  
char b[7];  
sprintf(b,"%f",v2);
```

```
struct datos *d = (struct datos*)malloc(sizeof(struct datos));  
d->v3=htons(30);  
d->v4="cadena";
```

```
if(sendto(cd, (const char*)msj, strlen(msj)+1, 0, (struct sockaddr *)rp->ai_addr, rp->ai_addrlen)==-1)
```

```
if(sendto(cd, &v1, sizeof(v1), 0, (struct sockaddr *)rp->ai_addr, rp->ai_addrlen)==-1)
```

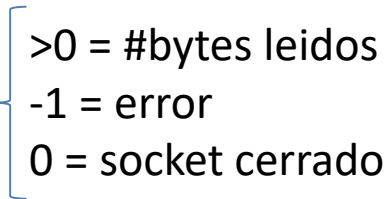
```
if(sendto(cd, b, strlen(b)+1, 0, (struct sockaddr *)rp->ai_addr, rp->ai_addrlen)==-1)
```

```
if(sendto(cd, (const char*)d, sizeof(d), 0, (struct sockaddr *)rp->ai_addr, rp->ai_addrlen)==-1)
```

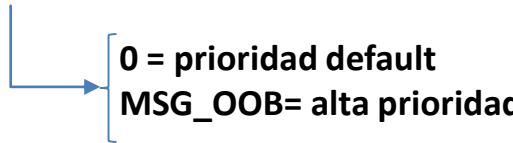
# Función recvfrom()

`#include <sys/socket.h>`

`ssize_t recvfrom(int sd, const void *buf, size_t tam, int bandera, const struct sockaddr *dst, socklen_t *tam)`

Valor devuelto: 

- >0 = #bytes leídos
- 1 = error
- 0 = socket cerrado



- 0 = prioridad default
- MSG\_OOB = alta prioridad

```
char buf[100];
int n = recv(cd, buf, sizeof(buf), 0);
if (n < 0)
    perror("Error en la función recv\n");
else if (n == 0) {
    perror("Socket cerrado\n");
    exit(1);
}
int v;
n = recv(cd, &v, sizeof(v), MSG_OOB);
...
```

# Ej. recvfrom()

```
char *m =(char *)malloc(sizeof(char)*20);  
memset(m,0,sizeof(m));
```

```
int v1;
```

```
float v2;  
char b[7];  
memset(b,0,sizeof(b));
```

```
struct datos *d;  
Char bb[20];
```

```
struct sockaddr_storage rp;  
memset(&rp,0,sizeof(rp));  
socklen_t ctam = sizeof(rp);
```

```
if(recvfrom(cd, m, sizeof(m), 0, (struct sockaddr *)&rp, &ctam)==-1)
```

```
if(recvfrom(cd, &v1, sizeof(v1), 0, (struct sockaddr *)&rp, &ctam)==-1)  
int vv = ntohl(v1);
```

```
if(recvfrom(cd, b, sizeof(b), 0, (struct sockaddr *)&rp, &ctam)==-1)  
v2 = atof(b);
```

```
if(recvfrom(cd, (const char*)bb, sizeof(bb), 0, (struct sockaddr *)&rp,&ctam)==-1)
```

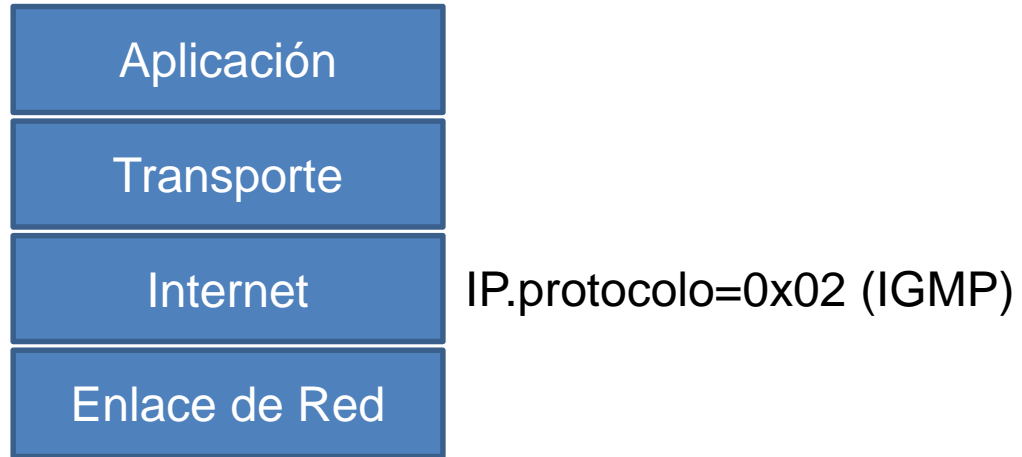
# Ejercicio

- Crear un programa que permita al usuario jugar el juego “Ahorcado” en red implementando el servidor en lenguaje C y el cliente en lenguaje JAVA.



# Sockets de datagrama multicast bloqueantes en C

# Internet Group Management Protocol (IGMP)



# Mensaje IGMP



Tipo {  
   $(0x11)_{16} = (17)_{10} \Rightarrow$ Consulta  
   $(0x12)_{16} = (18)_{10} \Rightarrow$ Reporte (IGMPv1)  
   $(0x16)_{16} = (22)_{10} \Rightarrow$ Reporte (IGMPv2)  
   $(0x22)_{16} = (34)_{10} \Rightarrow$ Reporte (IGMPv3)

Tiempo { Solo para el tipo  $(0x11)_{16}$  en milisegundos

# Opción de socket SO\_REUSEADDR

```
int op,v=1;  
if ( setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &v, sizeof(v)) != 0 )  
... perror("No se pudo modificar la opción \n ");
```

# Estructura ip\_mreq (ipv4)

```
struct ip_mreq {  
    struct in_addr imr_multiaddr; /* Dir. Grupo multicast */  
    struct in_addr imr_address; /* Dir. Interfaz de red local */  
};
```

# Estructura ipv6\_mreq (ipv6)

```
struct ipv6_mreq {  
    struct in6_addr  ipv6mr_multiaddr; /* Dir. IPv6 multicast */  
    unsigned int     ipv6mr_interface; /* índice interfaz red */  
}
```

ffxe::/16	224.0.1.0- 238.255.255.255	Alcance Global	
-----------	-------------------------------	----------------	--

# Opción de socket IP\_ADD\_MEMBERSHIP (IPv4)

```
struct ip_mreq mr;
/* Ponemos la dirección de grupo */
    memcpy(&mr.imr_multiaddr,&((struct sockaddr_in*)(maddr->ai_addr))-
>sin_addr,sizeof(mr.imr_multiaddr));

    /* Aceptamos datagramas multicast por cualquier interfaz */
    mr.imr_interface.s_addr = htonl(INADDR_ANY);

    /* Nos unimos a la dirección de grupo */
    if ( setsockopt(sd, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char*) &mr, sizeof(mr))
!= 0 )
    {
        perror("setsockopt() \n");
    }
```

# Opción de socket IP\_ADD\_MEMBERSHIP (IPv6)

```
struct ipv6_mreq mr; /* Multicast address join structure */

/* Especificamos la dirección de grupo IPv6 */
memcpy(&mr.ipv6mr_multiaddr, &((struct sockaddr_in6*)(maddr->ai_addr))-
>sin6_addr, sizeof(mr.ipv6mr_multiaddr));

/* Aceptamos datagramas multicast IPv6 desde cualquier interfaz de red */
mr.ipv6mr_interface = 0;

/* Nos unimos a la dirección de grupo */
if ( setsockopt(sd, IPPROTO_IPV6, IPV6_ADD_MEMBERSHIP, (char*) &mr,
sizeof(mr)) != 0 )
{
    perror("setsockopt() \n");
}
```



# Opción de socket SO\_REUSEADDR

```
Unsignet char ttl= 200;  
if ((setsockopt(sd, IPPROTO_IP, IP_MULTICAST_TTL,(void*) &ttl, sizeof(ttl))) < 0)  
    perror("setsockopt() \n");
```

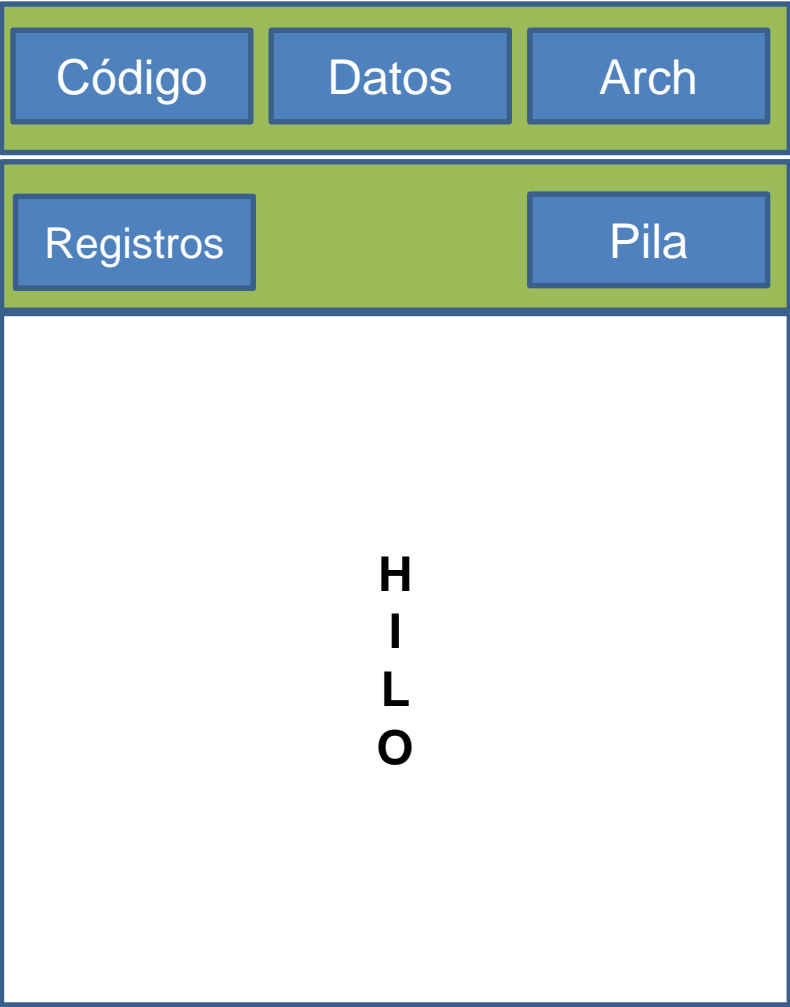
\*Ej. cliente2.c, servidor2.c

Hilos (threads)

# Proceso vs Hilo

<ul style="list-style-type: none"><li>-Tiene un estado: <b>Nuevo, Listo, Ejecución, Bloqueado, Finalizado</b></li><li>-Contador de programa</li><li>-Registros del CPU</li><li>-Información de planificación: <b>(prioridad, cola en que está agendado)</b></li><li>-PID</li><li>-Información de admón. de memoria(mapeo: páginas o segmentos)</li><li>-Pila</li><li>-Información de contabilidad (recursos utilizados)</li></ul>	<ul style="list-style-type: none"><li>-Tiene un estado: Ejecución, Listo, Bloqueado</li><li>-Registros del CPU(contexto)</li><li>-Pila</li><li>-Información de planificación</li><li>-Variables locales</li><li>-Identificador</li></ul>

# Proceso vs Hilo



# Creación de hilos en JAVA

Heredando de la clase Thread  
(`java.lang.Thread`)

Implementando la interfaz Runnable  
(`java.lang.Runnable`)

# Clase Thread (java.lang.Thread)

## **Campos:**

static int MAX\_PRIORITY

static int MIN\_PRIORITY

static int NORM\_PRIORITY

## **Constructores:**

Thread( )

Thread(String nombre)

Thread(ThreadGroup gpo, String n)

Thread(Runnable r)

Thread(Runnable r, String nombre)

Thread(ThreadGroup gpo, Runnable r)

# Métodos:

static int activeCount( )  
protected Object clone( )  
static Thread currentThread( )  
static void dumpStack( )  
long getId( )  
String getName( )  
int getPriority( )  
Thread.State getState( )  
ThreadGroup getThreadGroup( )  
static boolean holdsLock(Object o)  
boolean isAlive( )  
boolean isDaemon( )  
void join( )  
void join(long t)  
void join(run)

- void setDaemon(boolean b )
- void setName(String nombre)
- void setPriority(int p)
- static void sleep(long t)
- void start( )
- String toString( )
- Static void yield( )

# Interfaz Runnable (java.lang.Runnable)

## Métodos:

`void run( )`



# Situaciones al compartir recursos

**Condición de carrera:** Ocurre cuando dos o más hilos acceden al mismo tiempo a un recurso compartido, de modo que el resultado de este acceso depende del orden de llegada de los hilos.

**Sección crítica:** sección de un programa en que se accede a un recurso compartido, el cual no debe ser accedido por más de un hilo a la vez.

**Exclusión mutua:** Un solo hilo debe excluir temporalmente a los demás hilos para utilizar un recurso compartido.

# Sincronización de hilos

- A nivel de bloque
- A nivel de método
- A nivel de variable (visibilidad)
- Mutex
- Variables de condición
- Semáforos

# Sincronización a nivel de bloque

```
synchronized (Object o){  
  
    //...  
}
```

\*Ej. BloqueSinc.java

# Sincronización a nivel de método

```
public class Clase {  
    public synchronized void método(. . .) {  
    ... }  
    . . .  
}
```

\*Ej. MetodoSinc.java

# Sincronización a nivel de variable

```
class Contador{  
    private volatile int vcuenta;  
    public Contador(){  
        vcuenta=0;  
    }//constructor Contador
```

# Mutex (`java.util.concurrent.*;`)

- Interfaz Lock
- Clase ReentrantLock

# Interfaz Lock (java.util.concurrent.Lock)

Un bloqueo (lock) es un mecanismo para controlar el acceso a un recurso compartido por múltiples hilos.

```
Lock l = ...;  
l.lock();  
try {  
    // acceso al recurso compartido protegido por el bloqueo  
} finally {  
    l.unlock();  
}
```

# Interfaz Lock

## Métodos:

- void lock( );
- void lockInterruptibly( )
- Condition new Condition( )
- boolean tryLock( )
- boolean tryLock(long t, TimeUnit unidades)
- void unlock( )



# Clase ReentrantLock

(java.util.concurrent.locks.ReentrantLock)

## Constructores:

ReentrantLock( )

ReentrantLock(boolean justicia)

## Métodos:

protected Thread getOwner( )

int getHoldCount( ) // #veces lock sin unlock

int getQueueLength( )

protected Collection<Thread> getWaitingThreads(Condition c)

int getWaitingQueueLength(Condition c )

boolean isFair( )

boolean isLocked( )

void lock( )

void lockInterruptibly( )

# Clase ReentrantLock

(`java.util.concurrent.locks.ReentrantLock`)

- Métodos
  - `Condition newCondition( )`
  - `boolean tryLock( )`
  - `boolean tryLock(long t, TimeUnit unidades)`
  - `void unlock( )`

# Clase ReentrantLock

## (java.util.concurrent.locks.ReentrantLock)

Uso:

```
class miClase {  
    private final ReentrantLock rl = new ReentrantLock();  
    // ...  
  
    public void metodo() {  
        lock.lock(); // comienza mutex  
        try {  
            // ... Cuerpo del método  
        } finally {  
            lock.unlock()  
        }  
    }  
}
```

\*\*Ejemplo: Mutex.java

## 1.3.2 Sockets orientados a conexión no bloqueantes

# Socket bloqueante (1/2)

- Las entradas y salidas son por naturaleza bloqueantes (no permiten realizar nada mas hasta que terminen)
- En el caso de los sockets si no hay nada que procesar la instrucción se queda dormida hasta que ocurra un evento que permita terminar la operación

# Socket bloqueante (2/2)

- Si realizamos operaciones de entrada (*read, recv, recvfrom, etc.*) sobre el socket y no hay datos disponibles el proceso entrara al estado de dormido hasta que haya datos para leer
- Si realizamos operaciones de salida (*write, send, sendto, etc.*) sobre el socket, el kernel copia los datos del buffer de la aplicación en el buffer de envio de datos, si no hay espacio en este último el proceso se bloqueara hasta tener suficiente espacio

# Socket no bloqueante

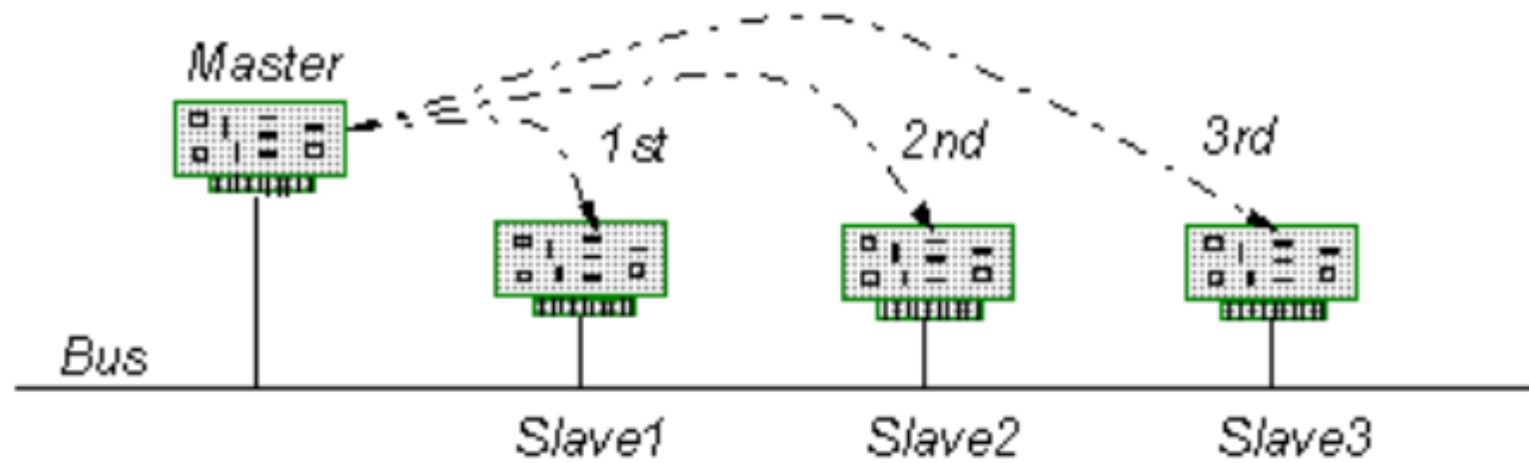
- En algunas ocasiones es preferible que no exista el bloqueo mencionado
- Permite realizar otras tareas si no hay datos que manejar
- Hay dos maneras básicas de manejo:
  - *Polling*
  - *Asíncrono*

# Polling

- Consiste en una operación de consulta constante
- Eso lo vuelve síncrono, ya que solo se procesa en un momento determinado



# Polling



# Asíncrono

- En este caso, hay que esperar a que ocurra un evento de entrada o salida y actuar en consecuencia

Sockets orientados a conexión no  
bloqueantes en java

# Dos tópicos

- *ServerSocketChannel*
- *Iterator*

# ServerSocketChannel

- Según la documentación oficial de Java un canal es:  
[...] representa una conexión abierta a una entidad como un dispositivo de hardware, un archivo, un socket de red o un componente de software que es capaz de realizar una o mas operaciones distintas de E/S; por ejemplo, leer o escribir. Un canal está abierto tras su creación, y una vez cerrado permanece cerrado. Una vez que un canal está cerrado, cualquier intento de llamar a una operación de E/S sobre él causará que se arroje una *CloseChannerException*.

# ServerSocketChannel

- Dicho de otra forma: un canal es una conexión entre un buffer y una fuente o un consumidor de datos.
- Los datos pueden leerse de los canales mediante buffer
- Los datos de un buffer se pueden escribirse en los canales

# ServerSocketChannel

- Los canales pueden funcionar con bloqueo o sin el
- Una operación de E/S con bloqueo no retorna hasta que se produce una de estas situaciones:
  - Se completa la operación
  - Se produce una interrupción debido al SO
  - Se lanza una exception
- Todos los métodos **read()** y **write()** de **java.io** producen bloqueo hasta que se produce alguno de los casos anteriores

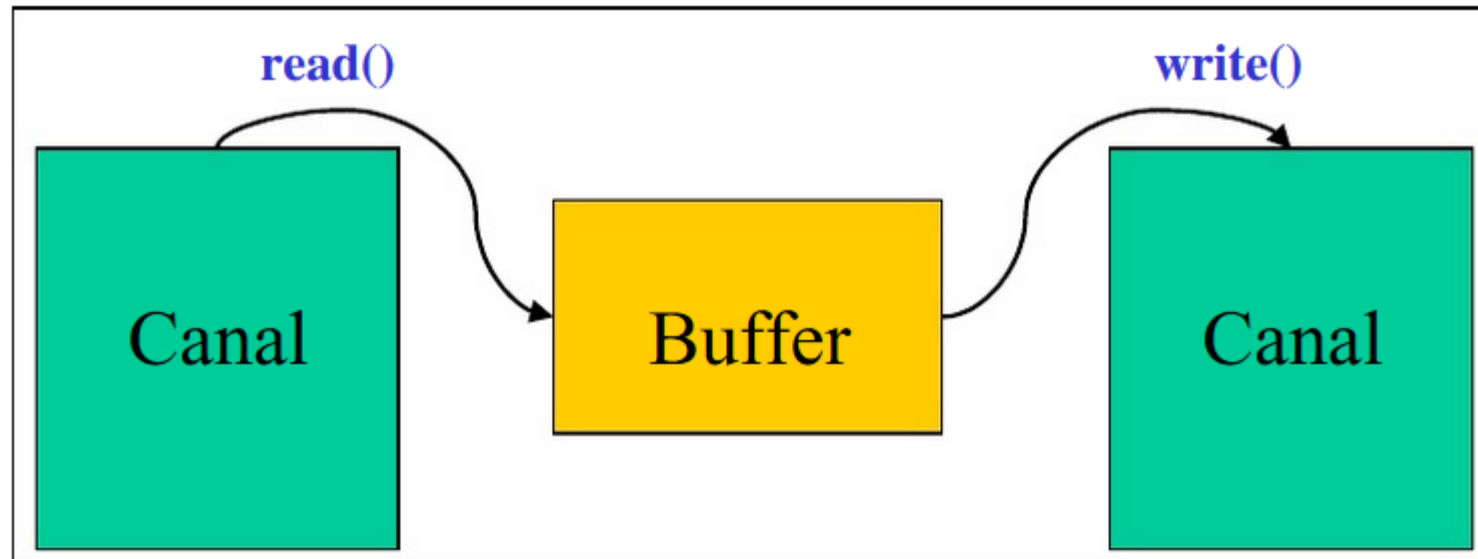
# ServerSocketChannel

- Una operación de E/S sin bloqueo retorna al instante, devolviendo algún valor de retorno que indique si la operación fue exitosa o no
- Un programa o hilo que ejecute una operación sin bloque no se quedara dormido esperando datos, una interrupción o una excepción; su curso de ejecución continuara.

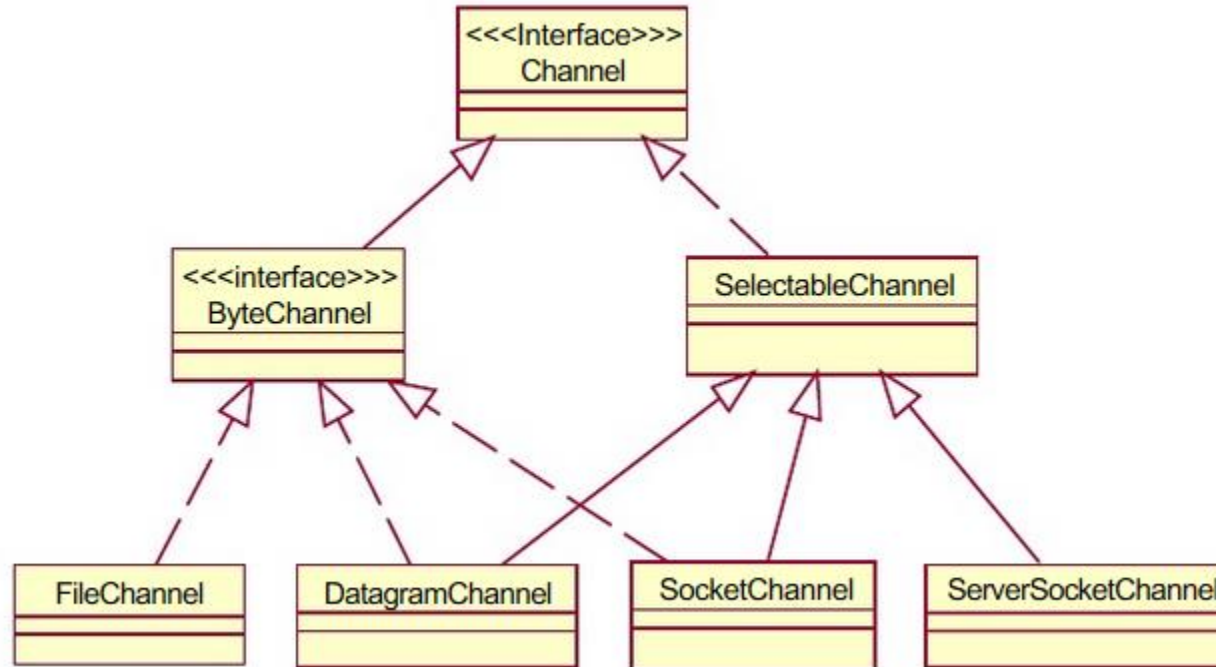


# ServerSocketChannel

- Los buffers son lo intermediarios entre canales
- No es posible pasar directamente los datos de un canal a otro



# Jerarquia simplificada de java.nio.channel



# Clase ServerSocketChannel

- La clase `java.nio.channels.ServerSocketChannel` es un canal seleccionable para sockets TCP pasivos
- Viene a ser un envoltorio para un objeto `ServerSocket`, al cual asocia un canal.
- Para crearlo hay que usar el método `public static ServerSocketChannel open() throws IOException`

# Clase ServerSocketChannel

- Un **ServerSocketChannel** recién creado no está ligado a ningún puerto
- La liga se consigue usando el método **bind()**
- El método **socket()** regresa el socket de servidor asociado al canal
- El método **accept()** acepta una conexión echa al socket del canal
- El método **configureBlocking(boolean  
blocking)**

# Ejemplo

```
ServerSocketChannel canalServidor =  
ServerSocketChannel.open();  
  
canalServidor.socket().bind(new  
InetSocketAddress("localhost",9000));  
canalServidor.configureBlocking(false);  
  
While(true){  
    SocketChannel canalSocket
```

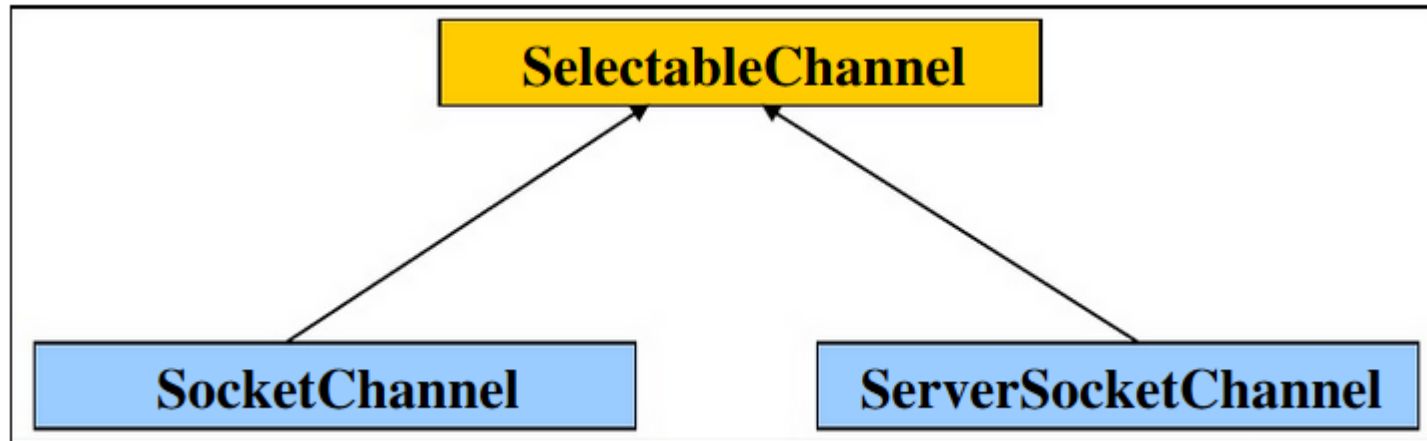
# Clase SocketChannel

- La clase `java.nio.channels.SocketChannel` es un canal seleccionable para sockets TCP activos
- Es un envoltorio para un objeto socket , que permite asociarlo a un canal
- Los objetos de esta clase se crean mediante llamadas al método estático `open()`
- De la misma forma que el canal anterior se puede definir como bloqueante o no bloqueante

# Ejemplo

```
//Se crea un objeto SocketChannel
SocketChannel canalSocket = SocketChannel.open();
//Se conecta usando un objeto InetAddress
canalSocket.connect(new InetAddress("localhost",9000));
//Se configura sin bloqueo
canalSocket.configureBlocking(false);
//Se crea un buffer y se lee del canal
ByteBuffer buffer = ByteBuffer.allocate(1024);
Buffer.clear();
canalSocket.read(buffer);
```

# SelectableChannel





# Selector y SelectorKey

- La clase `java.nio.channels.Selector` es una de las principales de la API NIO
- Un objeto *Selector* controla una serie de canales y lanza un aviso cuando uno de ellos lanza un suceso E/S
- La clase `Selector` informa a la aplicación de las operaciones de E/S que ocurren los canales que están activos

# Selector y SelectorKey

- La información sobre las operaciones de E/S se registra en un conjunto de claves, que son instancias de la clase *SelectorKey*
- Cada clave almacena información sobre el canal que desencadena la operación y el tipo de ella (lectura, escritura, conexión entrante, conexión aceptada)
- En la clase *Selector*, las instancias se crean con el método estático `open()`

# Método select()

- El método **select()** bloquea el programa hasta que algún canal recibe datos
- Su origen proviene de una llamada al sistema operativo UNIX, aunque mucho menos engorroso que en C
- Con una sola llamada a **select()** se espera simultáneamente a todas las entradas de los clientes

# Ejemplo

//Se obtiene una dirección de socket

```
InetSocketAddress dir = new InetSocketAddress("localhost",9000);
```

//Se crea el canal

```
ServerSocketChannel canalServer = ServerSocketChannel.open();
```

```
canalServer.configureBlocking(false);
```

```
canalServer.socket().bind(dir);
```

//Se crea un objeto Selector

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

//Se registra el canal con el selector para que esté al tanto de lo que ocurre

```
canalServer.register(selector, SelectionKey.OP_CONNECT | SelectionKey.OP_READ  
                        | SelectionKey.OP_WRITE);
```

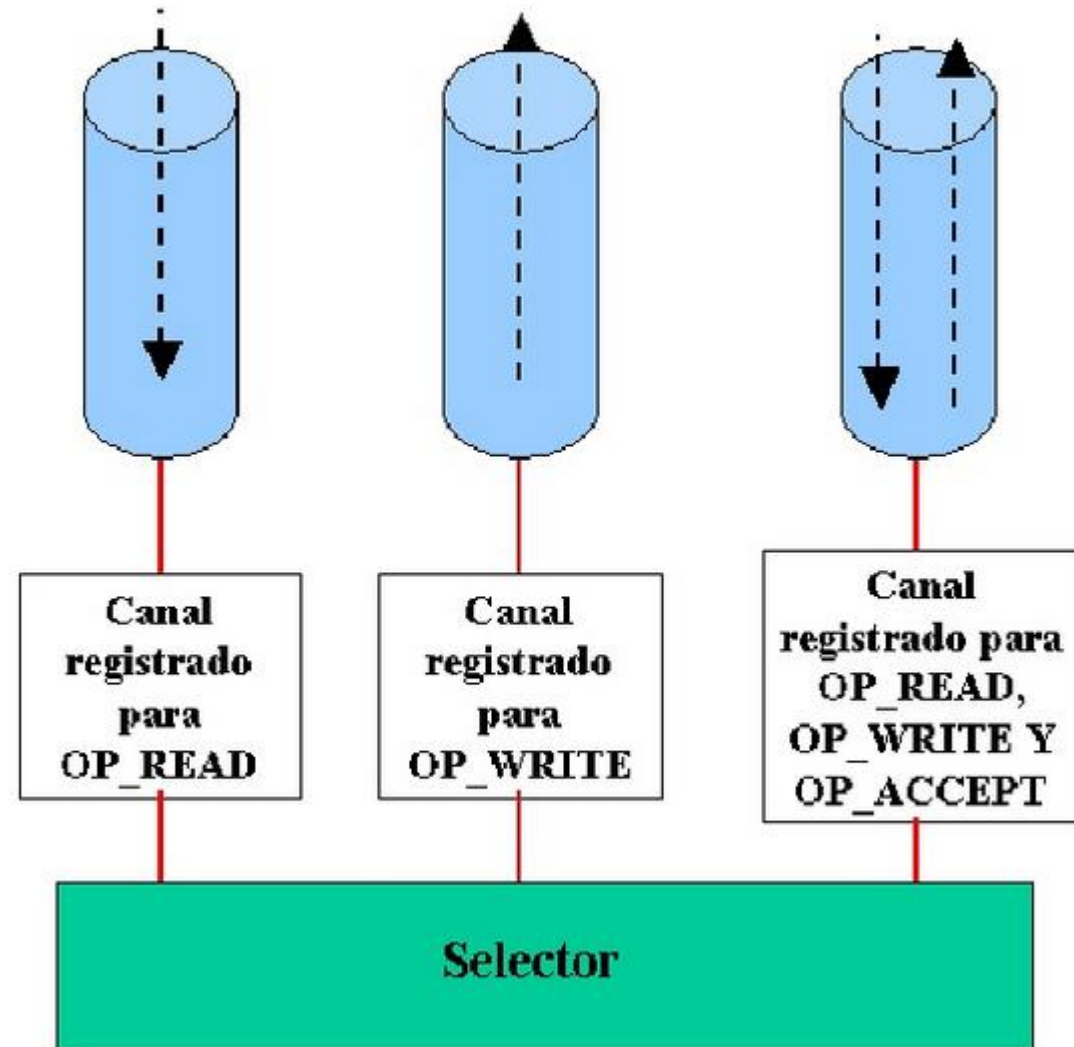
# Selector

- Para poder saber que suceso de E/S de los que estamos interesados ocurre, es necesario registrar el canal con un selector y especificar el tipo o tipos de sucesos de interés
- Una clase es seleccionable si puede registrarse con un selector
- Todos los canales que descenden de la clase *SelectableChannel* son seleccionables

# Sucesos de un selector

- Los sucesos que se pueden registrar mediante un selector se especifican con las siguientes constantes enteras:
  - `SelectionKey.OP_READ`
  - `SelectionKey.OP_WRITE`
  - `SelectionKey.OP_ACCEPT`
  - `SelectionKey.OP_CONNECT`
- Distintos canales pueden registrarse para diversos sucesos
- De ahí que se diga que un conjunto de canales se multiplexa con un selector

# Multiplexación de canales



# Interfaz Iterator

- La interfaz *Iterator* se encuentra en **java.lang**
- Implementar *Iterable* tan solo obliga a sobrescribir el método **iterator()**
- *Iterator* es un tipo abstracto definido por la interfaz *List*
- No puede ser instanciado porque carece de constructor
- Se puede definir un objeto si se instancia en una clase que implemente la interface



# Ejemplo

- Erroneo:

```
Iterator <SelectionKey> selecciones = new Iterator <SelectionKey>();
```

- Correcto:

```
Iterator <SelectionKey>it = selector.selectedKey().iterator();
```

Sockets orientados a conexión no  
bloqueantes en C

# fcntl() y el polling

- **fcntl()** es una función que permite realizar diferentes operaciones sobre descriptores

- El prototipo de la función es:

```
#include <fcntl.h>
```

```
int fcntl(int fd, int cmd, /*int arg*/)
```

# fcntl() y el polling

- Cada descriptor tiene asociado una serie de banderas que nos permite obtener información de dicho descriptor
- Para obtener el valor de las banderas se hace una llamada a **fcntl()** con el segundo parámetro con valor **F\_GETFL**
- Si queremos modificar el valor de una bandera se pasa como segundo parámetro **F\_SETFL**

# fcntl() y el polling

- Para hacer que un socket sea no bloqueante hay que activar la bandera **O\_NONBLOCK**

```
if(fcntl(sd, F_SETFL, O_NONBLOCK) < 0)
```

```
    perror("fcntl: no se puede fijar operación no bloqueante");
```

- Ahora, cuando se realice una operación de lectura o escritura en un socket y no se pueda completar, la función regresará un valor de -1 y se asignará el valor de **EWOULDBLOCK** a la variable *errno*

# A sincronía usando señales

- La señal *SIGIO* se genera cuando cambia el estado de un socket, por ejemplo:

- Existen nuevos datos en el buffer o se ha liberado espacio
- Hay nuevas solicitudes de conexión

- Para que el socket genere la señal SIGIO debemos de modificar la bandera **O\_ASYNC**

```
if(fcntl(sd, F_SETFL, O_ASYNC | O_NONBLOCK) < 0)
    perror("Error en el fcntl");
```

- Los mecanismos asíncronos utilizan esta señal para saber cuando están listos los datos en un socket

# select()

- Esta función te permite comprobar varios sockets al mismo tiempo
- Te da información sobre los sockets del tipo:
  - Listo para leer
  - Listo para escribir
  - A ocurrido una excepción
- La función comprueba conjuntos de descriptores de fichero

# Prototipo de select

- Prototipo:

```
#include <sys/time.h>
```

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
int select(int numfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout)
```

- Donde:

- numfds* debe de tener el mayor número de descriptor mas uno
- readfds* es el conjunto de descriptors de lectura
- writefds* es el conjunto de descriptors de escritura
- exceptfds* es el conjunto de descriptors de excepciones



# select()

- Si se quiere saber si se puede leer desde un socket se agrega este al conjunto *readfds*, si se quiere saber si se escribió en un socket se agrega a *writefds* y de igual forma con las excepciones usando *exceptfds*
- Para poder hacerlo se usan las siguientes macros:
  - **FD\_ZERO (fd\_set \*set);** borra un conjunto de descriptors
  - **FD\_SET(int fd, fd\_set \*set);** agrega un descriptor a un conjunto
  - **FD\_CLR(int fd, fd\_set \*set);** borra un descriptor de un conjunto
  - **FD\_ISSET(int fd, fd\_set \*set);** pregunta si *fd* está en un conjunto

# struct timeval

- Es una estructura de tiempo que permite establecer un periodo máximo de espera
- La estructura tiene los campos:

```
struct timeval {  
    int tv_sec; // Número de segundos  
    int tv_usec; // Número de microsegundo  
}
```