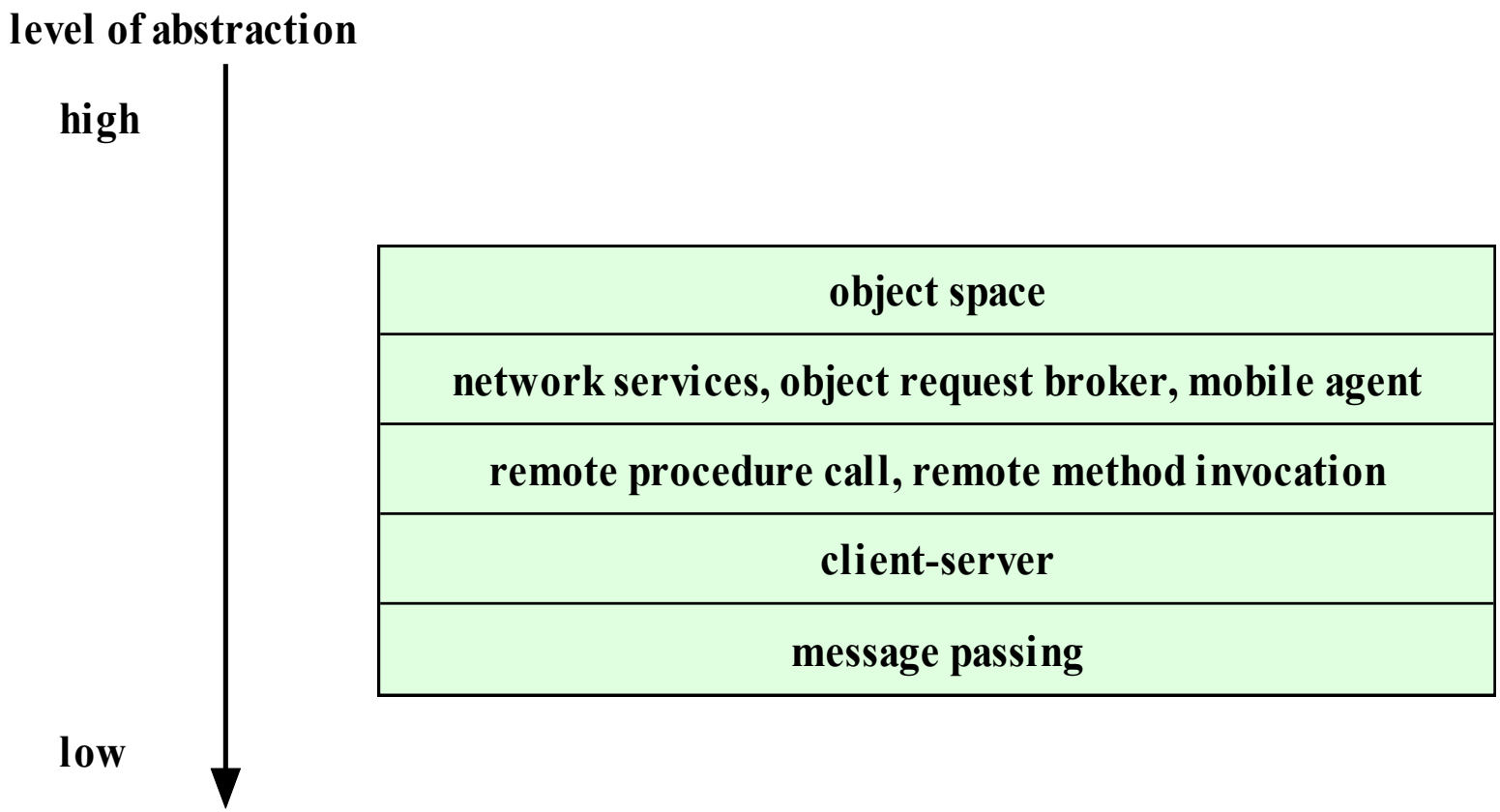
Software Distribuït - T4 - Client/Servidor

Eloi Puertas i Prats

Universitat de Barcelona
Grau en Enginyeria Informàtica

21 de febrer de 2024

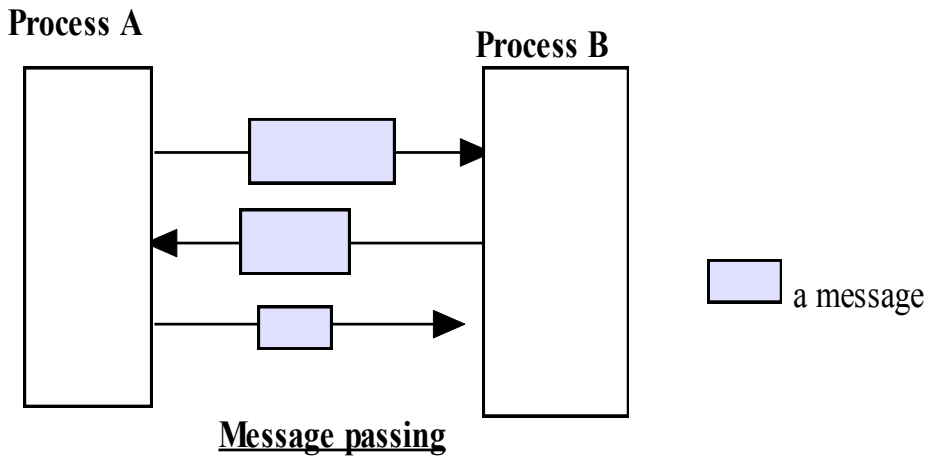
Paradigmes Aplicacions Distribuïdes



Pas de Missatges

El pas de missatges és el paradigma més bàsic per a les aplicacions distribuïdes:

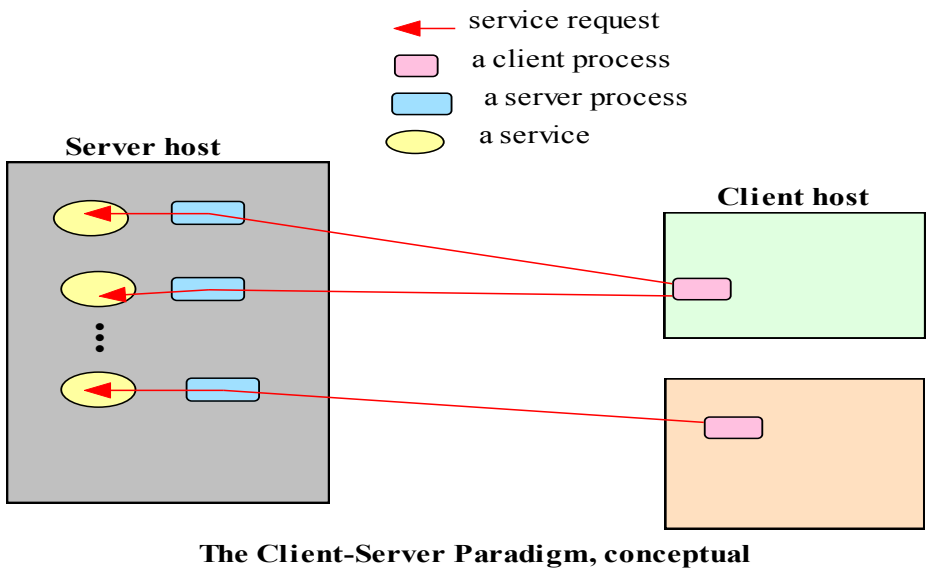
1. Un procés envia un missatge que representa una petició.
2. El missatge es lliura a un receptor, que processa la sol.licitud, i envia un missatge en resposta.
3. Al seu torn, la resposta pot desencadenar una nova sol.licitud, la qual condueix a una resposta posterior, i així successivament.



Client Servidor

El model client-servidor assigna rols asimètrics a dos processos que col.laboren

- Un procés, el *servidor*, fa el paper d'un proveïdor de serveis que espera passivament a l'arribada de les sol.licituds.
- L'altre procés, el *client*, fa sol.licituds específiques al servidor i espera la resposta.



The Client-Server Paradigm, conceptual

Client Servidor

Simple en el concepte, el model client-servidor proporciona una abstracció eficient per a la prestació de serveis de xarxa. Mitjançant l'assignació de rols asimètrics als dos costats, la sincronització d'esdeveniments s'ha simplificat:

- el procés del *servidor* espera peticions
- el *client* al seu torn, espera les respostes.

Entre les operacions que són requerides es troben:

- per part del *servidor* escoltar i acceptar les sol.licituds,
- per part d'un procés *client*, emetre peticions i rebre respostes.

Molts dels serveis d'Internet són aplicacions client-servidor. Aquests serveis es coneixen sovint pel protocol que l'aplicació implementa. Ben coneguts són els serveis d'Internet HTTP, FTP, DNS, Finger, Gopher, etc...

Operacions bàsiques

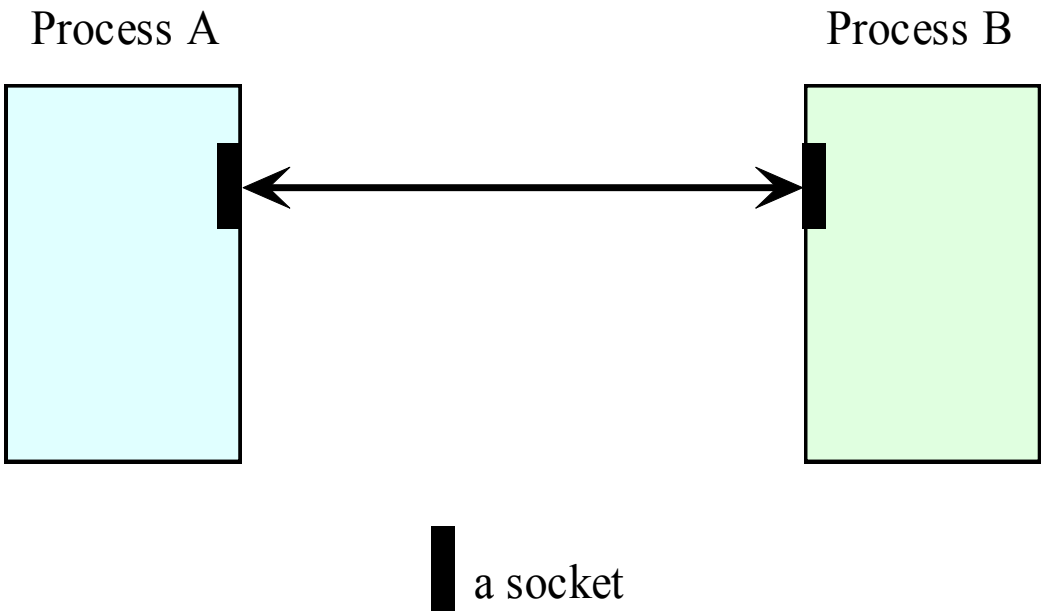
- Les operacions bàsiques que es requereixen per donar suport tant a pas de missatge com client-servidor són **enviar** i **rebre**.
- Per comunicacions orientades a la connexió, les operacions de **connexió** i **desconnexió** també es requereixen.
- Amb l'abstracció proporcionada per aquest model, els processos interconnectats realitzen operacions d'entrada/sortida d'una manera similar a E/S amb fitxers. Les operacions d'E/S encapsulen el detall de la comunicació de xarxa a nivell de sistema operatiu.
- L'API de sockets es basa en aquest paradigma: API Sockets

Sockets

L'API de sockets ofereix al programador un mecanisme de comunicació entre dos processos que poden residir en diferents màquines.
Un *socket* és punt final d'un enllaç de comunicació entre dos processos.
L'API està dissenyada per:

- Acomodar múltiples protocols de comunicació (TCP / IP, UNIX, APPLETALK)
- Implementar codi servidor que queda a l'espera de connexions i codi client que inicia aquestes connexions
- Ser coherent amb l'ús de fitxers en Unix.

Model conceptual de l'API de sockets

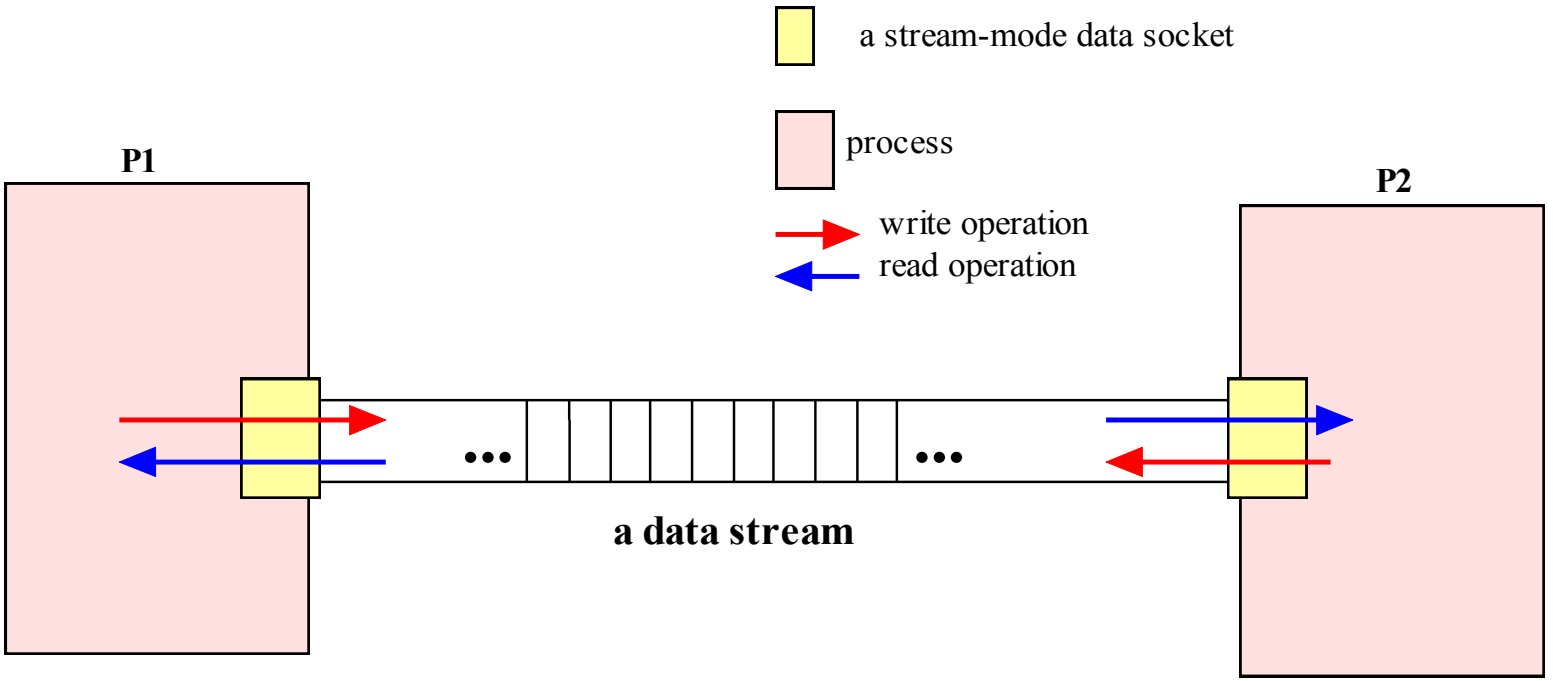


Tipus de sockets

Usualment els S.O. suporten tres tipus de sockets:

- Datagram -*datagrama*- (*SOCK_DATAGRAM*): Permeten als processos comunicar-se utilitzant UDP. Un socket Datagram proveeix un flux de missatges bidireccional no es garanteix duplicació o seqüencialitat. Es pot usar tant en comunicacions **unicast** com **multicast**
- Stream -*de flux*- (*SOCK_STREAM*): Permeten als processos comunicar-se utilitzant TCP. Proveeixen un flux de dades bidireccional, fiable, seqüencial, sense duplicació d'informació.
- Raw -*cru*- (*SOCK_RAW*). Permeten als processos comunicar-se utilitzant ICMP. Normalment són orientats a datagrama. Només el root pot usar-los i serveixen per desenvolupar protocols

Sockets Stream-mode

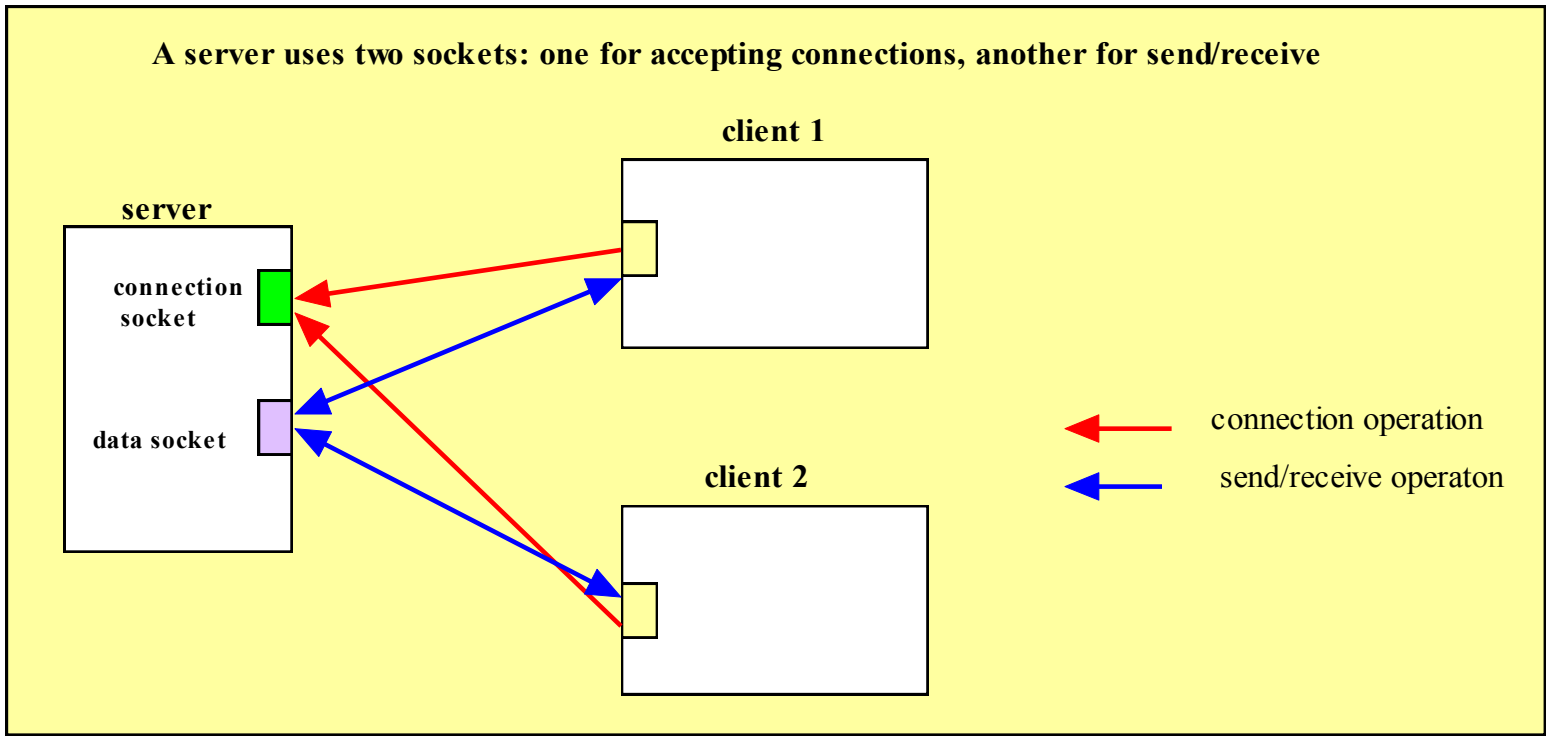


API de sockets Stream-mode

En Java, l'API de sockets stream-mode proveix dues classes:

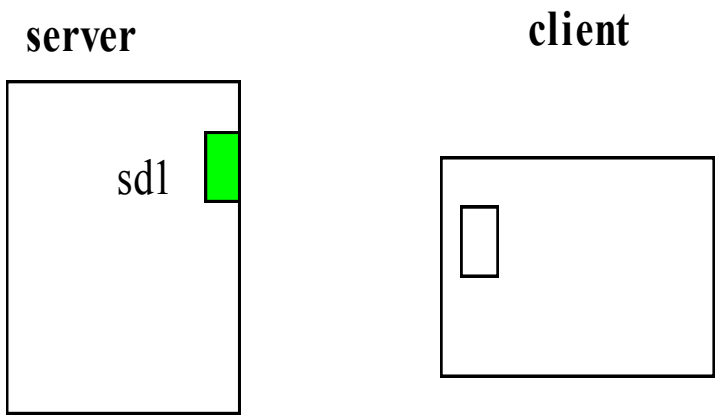
- ServerSocket: per acceptar connexions; un objecte d'aquesta classe n'hi direm *socket de connexió*.
- Socket: per intercanviar dades; un objecte d'aquesta classe n'hi direm un *socket de dades*.

Diagrama d'una connexió entre processos amb sockets orientats a la connexió



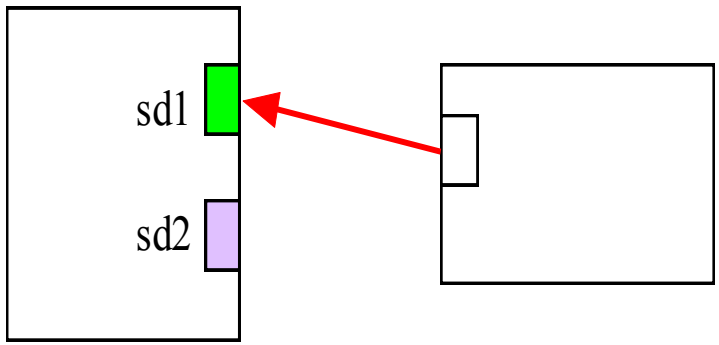
Sockets orientats a la connexio

1. Server establishes a socket sd1 with local address, then listens for incoming connection on sd1



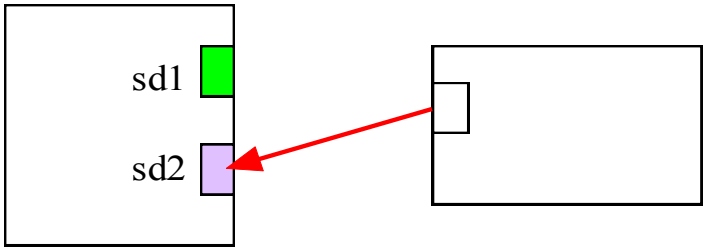
Client establishes a socket with remote (server's) address.

2. Server accepts the connection request and creates a new socket sd2 as a result.



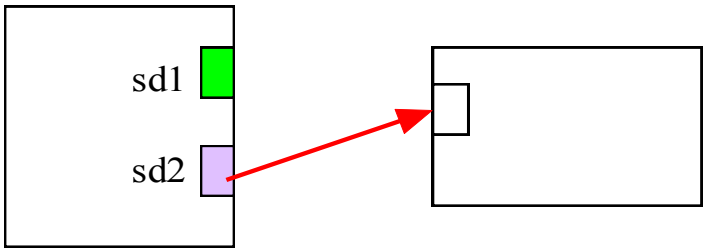
Sockets orientats a la connexio

3. Server issues receive operation using sd2.

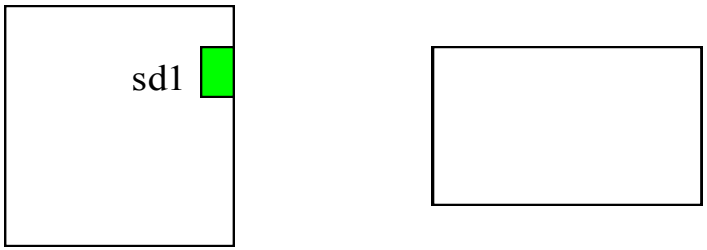


Client issues send operation.

4. Server sends response using sd2.



5. When the protocol has completed, server closes sd2; sd1 is used to accept the next connection



Client closes its socket when the protocol has completed

Diagrama de fluxe entre Servidor i Client

Server

1. **Create** a connection Socket
2. **Listen** for connection requests
3. **while** (true):
    4. **Accept** a connection
    5. **Creates** a data socket for I/O stream socket
    6. **Get** an InputStream for reading
    7. **Get** an OuputStream for writing
    8. **while** (not end):
        Do the protocol

finally! **Close** the data socket

finally! **Close** the connection socket

Client

1. **Create** a data socket
2. Request for a **Connection**
3. **Get** an InputStream for reading
4. **Get** an OuputStream for writing
5. **while** (not end):
    6. **Do** the protocol

finally! **Close** the data socket

UNIVERSITAT DE BARCELONA

Mètodes de l'API ServerSocket

Method/constructor	Description
ServerSocket(int port)	Creates a server socket on a specified port.
Socket accept() throws IOException	Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.
public void close() throws IOException	Closes this socket.
void setSoTimeout(int timeout) throws SocketException	Set a timeout period (in milliseconds) so that a call to accept( ) for this socket will block for only this amount of time. If the timeout expires, a java.io.InterruptedIOException is raised

Mètodes de l'API Socket

Method/constructor	Description
Socket(InetAddress address, int port)	Creates a stream socket and connects it to the specified port number at the specified IP address
void close() throws IOException	Closes this socket.
InputStream getInputStream( ) throws IOException	Returns an input stream so that data may be read from this socket.
OutputStream getOutputStream( )throws IOException	Returns an output stream so that data may be written to this socket.
void **setSoTimeout**(int timeout) throws SocketException	Set a timeout period for blocking so that a read( ) call on the InputStream associated with this Socket will block for only this amount of time. If the timeout expires, a **java.io.InterruptedIOException** is raised

I/O JAVA baix nivell Gestió d'enviaments parcials

Les funcions read no garanteixen la recepció de tots els caràcters que se'ls sol·licita. Això s'ha de gestionar.

```
public class Socket {
    public InputStream getInputStream()
}
public class InputStream{
    public int read(byte[] b)
}
```

- Es bloqueja fins que hi hagi dades disponibles.
- Llegeix un nombre indeterminat de bytes i els guarda al buffer **b**.
- Com a molt llegeix el n. bytes = a la longitud del buffer **b**.
- Retorna el nombre de bytes realment llegits.
- Read() i getInputStream() poden llençar una **IOException**, socket tancat, no connectat...

Exemples de tipus de dades

1. Escriure-Llegir sencers de 4 bytes per la xarxa. Ordre del bytes: BigEndian.
   - Transformar sencer a 4 bytes amb ordre Be.
   - Transformar 4 bytes amb ordre Be a sencer.
   - DataOutputStream: writeInt
2. Escriure-Llegir cadenes de càracters de mida fixa *x*
   - Transformar un String de longitud *x* a *x* chars d'un byte. (ignorar byte més alt)
   - Transformar *x* chars d'un byte a un string de longitud *x*.
   - DataOutputStream: writeBytes(string).
3. Escriure-Llegir cadenes de caràcters variables que codifiquen la seva longitud.
   - Determinar longitud de la capçalera
   - Escriure-Llegir el nombre de caràcters a/des de la capçalera
   - Escriure-Llegir el nombre de caràcters especificat.

Implementacions d'exemple a ComUtils: Sencers

```
import ...
public class ComUtils
{
   /* Objectes per escriure i llegir dades */
  private DataInputStream dis;
  private DataOutputStream dos;
  public ComUtils(Socket socket) throws IOException
  {
    dis = new DataInputStream(socket.getInputStream());
    dos = new DataOutputStream(socket.getOutputStream());
  }
    /* Llegir un enter de 32 bits */
  public int read_int32() throws IOException
  {
    byte bytes[] = new byte[4];
    bytes  = read_bytes(4);
    return bytesToInt32(bytes,"be");
  }
  /* Escriure un enter de 32 bits */
  public void write_int32(int number) throws IOException
  {
    byte bytes[]=new byte[4];
    int32ToBytes(number,bytes,"be");
    dos.write(bytes, 0, 4);
  }
}
```

Implementacions d'exemple a ComUtils: Llegir Bytes amb gestió enviament parcial

```
public class ComUtils
{...

        private byte[] read_bytes(int numBytes) throws IOException{
                int len=0 ;
                byte bStr[] = new byte[numBytes];
                do {
                        len += dis.read(bStr, len, numBytes-len);
                } while (len < numBytes);
                return bStr;
        }
}
```

Implementacions d'exemple a ComUtils: Transformacions

```
public class ComUtils
{...
        /* Passar d'enters a bytes */
        private int int32ToBytes(int number,byte bytes[], String endianess)
        {  if("be".equals(endianess.toLowerCase()))
                {   bytes[0] = (byte)((number >> 24) & 0xFF);
                        bytes[1] = (byte)((number >> 16) & 0xFF);
                        bytes[2] = (byte)((number >> 8) & 0xFF);
                        bytes[3] = (byte)(number & 0xFF);
                } else{
                        bytes[0] = (byte)(number & 0xFF);
                        bytes[1] = (byte)((number >> 8) & 0xFF);
                        bytes[2] = (byte)((number >> 16) & 0xFF);
                        bytes[3] = (byte)((number >> 24) & 0xFF);
                }return 4;}
        /* Passar de bytes a enters */
        private int bytesToInt32(byte bytes[], String endianess)
        {   int number;
                if("be".equals(endianess.toLowerCase()))
                { number=((bytes[0] & 0xFF) << 24) | ((bytes[1] & 0xFF) << 16) |
                        ((bytes[2] & 0xFF) << 8) | (bytes[3] & 0xFF);}
                else{number=(bytes[0] & 0xFF) | ((bytes[1] & 0xFF) << 8) |
                        ((bytes[2] & 0xFF) << 16) | ((bytes[3] & 0xFF) << 24);}
                return number;}
}
```

Implementacions d'exemple a ComUtils: Strings mida fixe

```
import ...
public class ComUtils
{       /* Llegir un string de mida STRSIZE */
        public String read_string() throws IOException
        {       String str;
                byte bStr[] = new byte[STRSIZE];
                char cStr[] = new char[STRSIZE];
                bStr = read_bytes(STRSIZE);
                for(int i = 0; i < STRSIZE;i++)
                        cStr[i]= (char) bStr[i];
                str = String.valueOf(cStr);
                return str.trim();
        }
        /* Escriure un string */
        public void write_string(String str) throws IOException
        {   int numBytes, lenStr;
                byte bStr[] = new byte[STRSIZE];
                lenStr = str.length();
                if (lenStr > STRSIZE){numBytes = STRSIZE;}
                else { numBytes = lenStr;}
                for(int i = 0; i < numBytes; i++){
                        bStr[i] = (byte) str.charAt(i);}
                for(int i = numBytes; i < STRSIZE; i++){
                        bStr[i] = (byte) '■';}
                dos.write(bStr, 0,STRSIZE);
        }
}
```

Implementacions d'exemple a ComUtils: Llegir Strings mida variable

```
public class ComUtils
{...

        /* Llegir un string  mida variable size = nombre de bytes especifica la longitud*/
        public  String read_string_variable(int size) throws IOException
        {
                byte bHeader[]=new byte[size];
                char cHeader[]=new char[size];
                int numBytes=0;
                // Llegim els bytes que indiquen la mida de l'string
                bHeader = read_bytes(size);
                // La mida de l'string ve en format text, per tant creem un string i el parsejem
                for(int i=0;i<size;i++){
                        cHeader[i]=(char)bHeader[i]; }
                numBytes=Integer.parseInt(new String(cHeader));
                // Llegim l'string
                byte bStr[]=new byte[numBytes];
                char cStr[]=new char[numBytes];
                bStr = read_bytes(numBytes);
                for(int i=0;i<numBytes;i++){
                        cStr[i]=(char)bStr[i];}
                return String.valueOf(cStr);
        }
}
```

UNIVERSITAT DE BARCELONA

Implementacions d'exemple a ComUtils: Escriure Strings mida variable

```
public class ComUtils
{...
        public  void write_string_variable(int size,String str) throws IOException
        {
                // Creem una üèseqncia amb la mida
                byte bHeader[]=new byte[size];
                String strHeader;
                int numBytes=0;
                // Creem la çcapalera amb el nombre de bytes que codifiquen la mida
                numBytes=str.length();
                strHeader=String.valueOf(numBytes);
            int len;
                if ((len=strHeader.length()) < size)
                for (int i =len; i< size;i++){
                        strHeader= "0"+strHeader;}
            System.out.println(strHeader);
                for(int i=0;i<size;i++)
                        bHeader[i]=(byte)strHeader.charAt(i);
                // Enviem la çcapalera
                dos.write(bHeader, 0, size);
                // Enviem l'string writeBytes de DataOutputStrem
                // no envia el byte éms alt dels chars.
                dos.writeBytes(str);
        }
}
```

Datagram-Mode Sockets

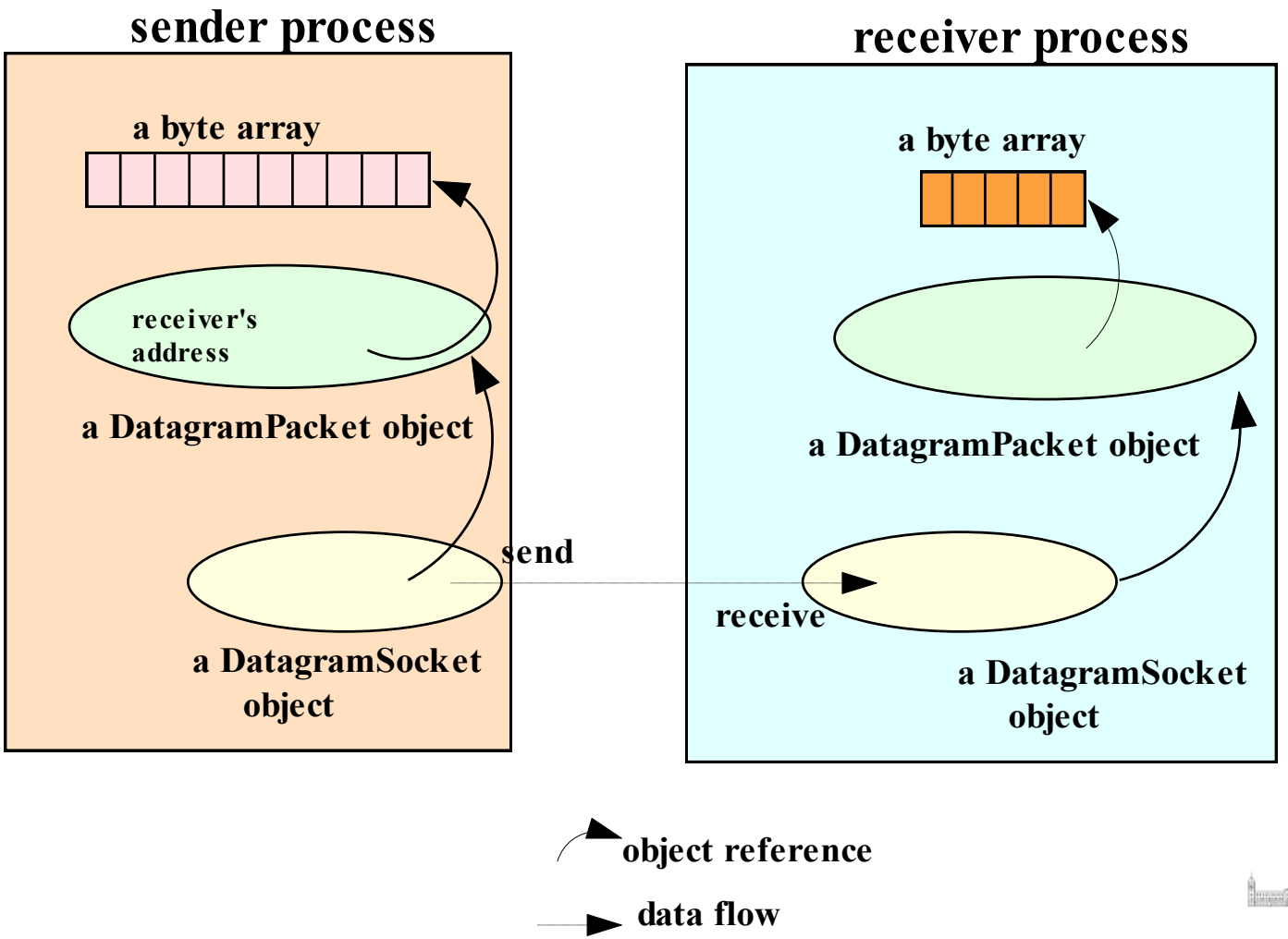


Diagrama de flux comunicació no orientada a la connexió de dos processos

Si el datagama és enviat abans de que l'operació de rebre sigui llançada, les dades seran descartades pel sistema i no seran rebudes.

Client

1. **Create** a datagram socket
2. **Bind** it to any local port
3. **Create** a datagram packet with data array and receiver address
4. invoke socket **Send** with the datagram packet

Server

1. **Create** a datagram socket
2. **Bind** it to any local port
3. **Create** a a datagram packet with data array
4. invoke socket **Receive** with the datagram packet

API de sockets Datagram-mode

En Java, l'API de sockets Datagram-mode proveix dues classes:

- DatagramSocket: per als sockets.
- DatagramPacket: per intercanviar datagrames.

Un procés que vulgui enviar o rebre dades fent servir aquesta API ha de crear una instància d'un objecte DatagramSocket. Cada socket està enllaçat a un port UDP de l'equip local.

Mètodes de l'API de Datagrames

Method/Constructor	Description
DatagramPacket(byte[ ] buf, int length)	Construct a datagram packet for receiving packets of length *length;* data received will be stored in the byte array reference by *buf*.
DatagramPacket (byte[ ] buf, int length, InetAddress address, int port)	Construct a datagram packet for sending packets of length *length* to the socket bound to the specified port number on the specified host *;* data received will be stored in the byte array reference by *buf*.
DatagramSocket ( )	Construct a datagram socket and binds it to any available port on the local host machine; this constructor can be used for a process that sends data and does not need to receive data.
DatagramSocket (int port)	Construct a datagram socket and binds it to the specified port on the local host machine; the port number can then be specified in a datagram packet sent by a sender.
void close( )	Close this datagramSocket object
void receive(DatagramPacket p)	Receive a datagram packet using this socket.
void send (DatagramPacket p)	Send a datagram packet using this socket.
void setSoTimeout (int timeout)	Set a timeout for the blocking receive from this socket, in milliseconds.

Sockets no orientats a la connexió

Amb sockets sense connexió, és possible que diversos processos enviïn simultàniament datagrames al mateix socket establert per un procés de recepció, en aquest cas l'ordre de l'arribada d'aquests missatges serà impredictible, d'acord amb el protocol UDP.

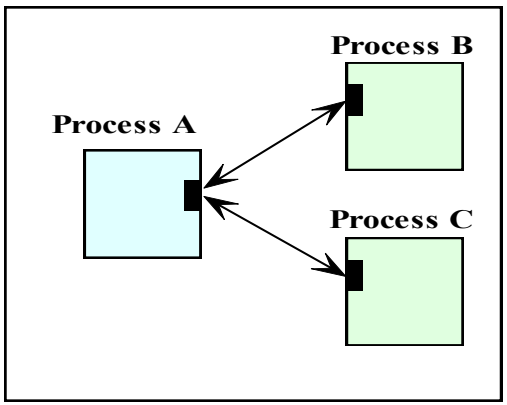


Figure 3a

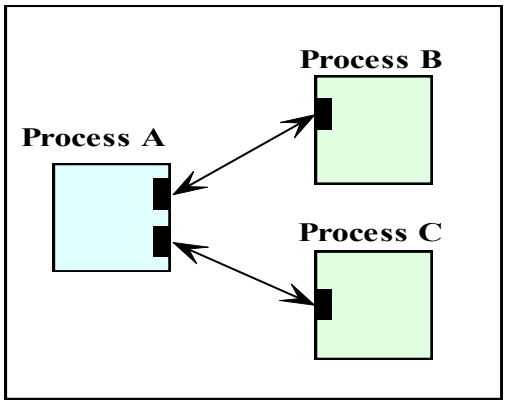


Figure 3b

a connectionless datagram socket

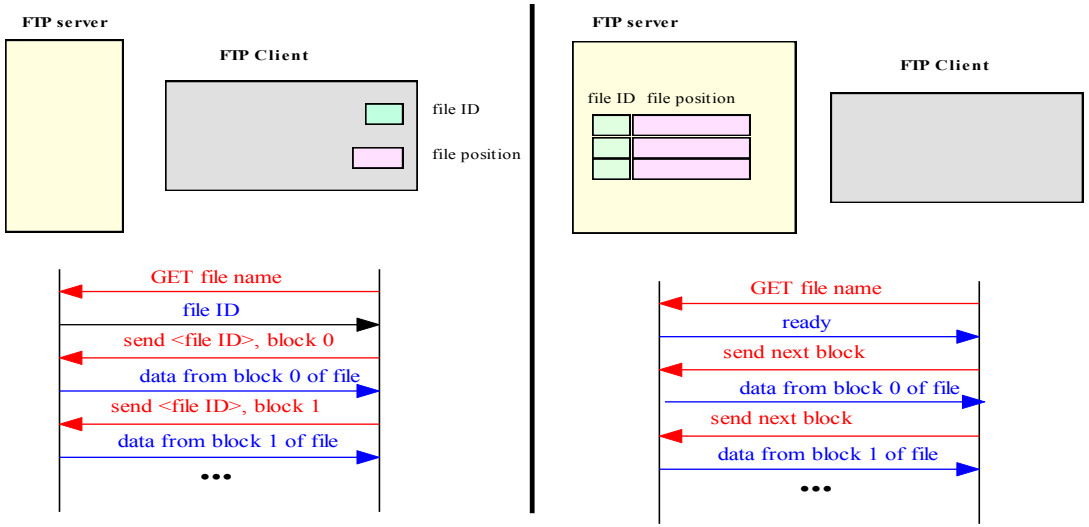
Tipus de servidors

Els servidors els podem categoritzar segons:

Amb estat	Vs	Sense estat.
Orientats a la connexió	Vs	No orientats a la connexió.
Iteratius	Vs	Concurrents.

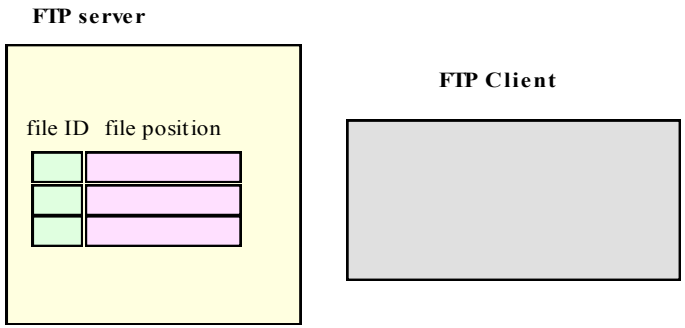
Amb estat vs Sense estat

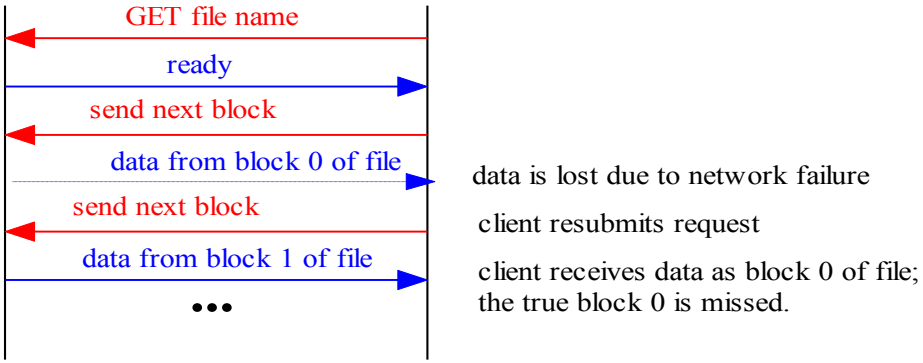
- Un servidor amb estat manté la informació de l'estat de cada client actiu.
- El fet de mantenir aquesta informació pot reduir la quantitat de dades intercanviades i per tant, el temps de resposta.
- Codificar un servidor sense estat és més senzill.



Amb estat vs Sense estat

- Mantenir la informació d'estat és dificultós amb presència d'errors.



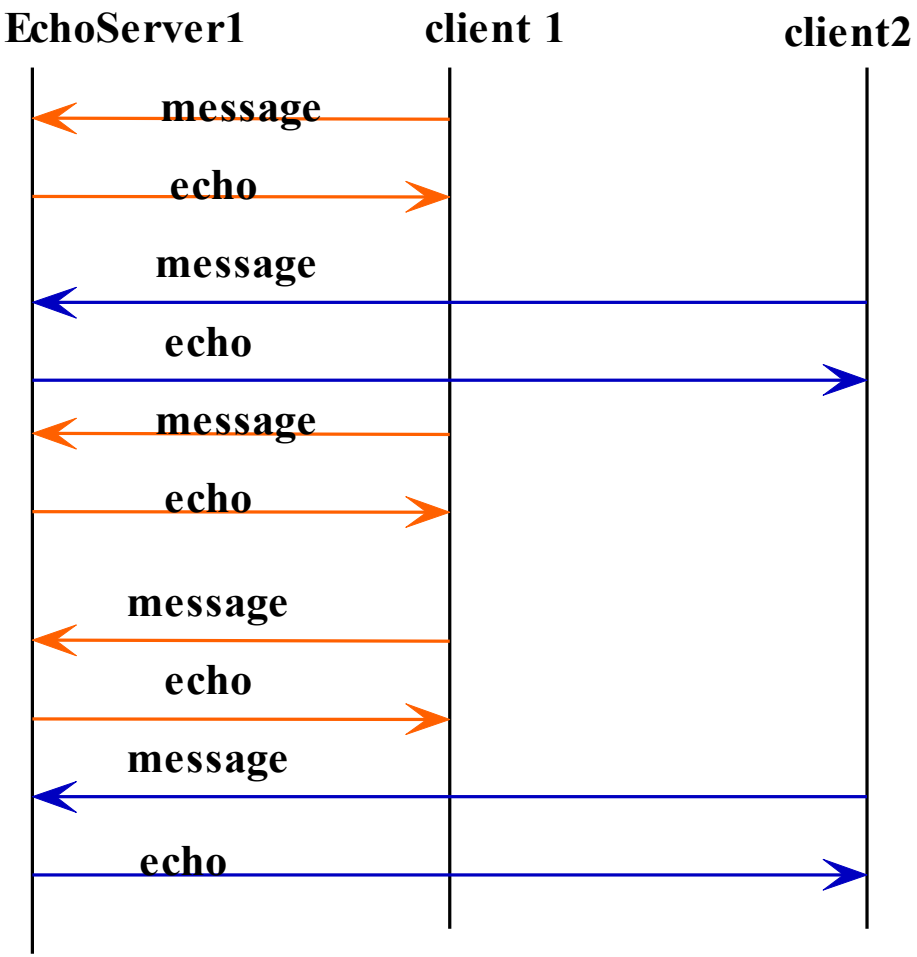


Orientats a la connexió Vs No orientats a la connexió

No orientats a la connexió Utilitza API Datagram-mode. Les sessions amb clients simultanis es poden intercalar.

Orientat a la connexió Utilitza API Stream-mode.

Diagrama d'esdeveniments de Servidor No Orientat a la connexió



Exemple Client/Servidor No Orientat a la connexió amb java

Exemple Client/Servidor UDP amb java

Iteratius vs Concurrents

Servidor Concurrent Fa servir diferents fils d'execució per donar servei a diversos clients alhora. Pot fer servir Threads, Processos o una convinació de tots dos.

Servidor iteratiu Només fa servir un sol fil d'execució. Les sessions amb clients simultanis les pot fer o seqüencialment o fent servir operacions no-bloquejants (usant selector).

Servidor orientat a la connexió Concurrent amb Threads

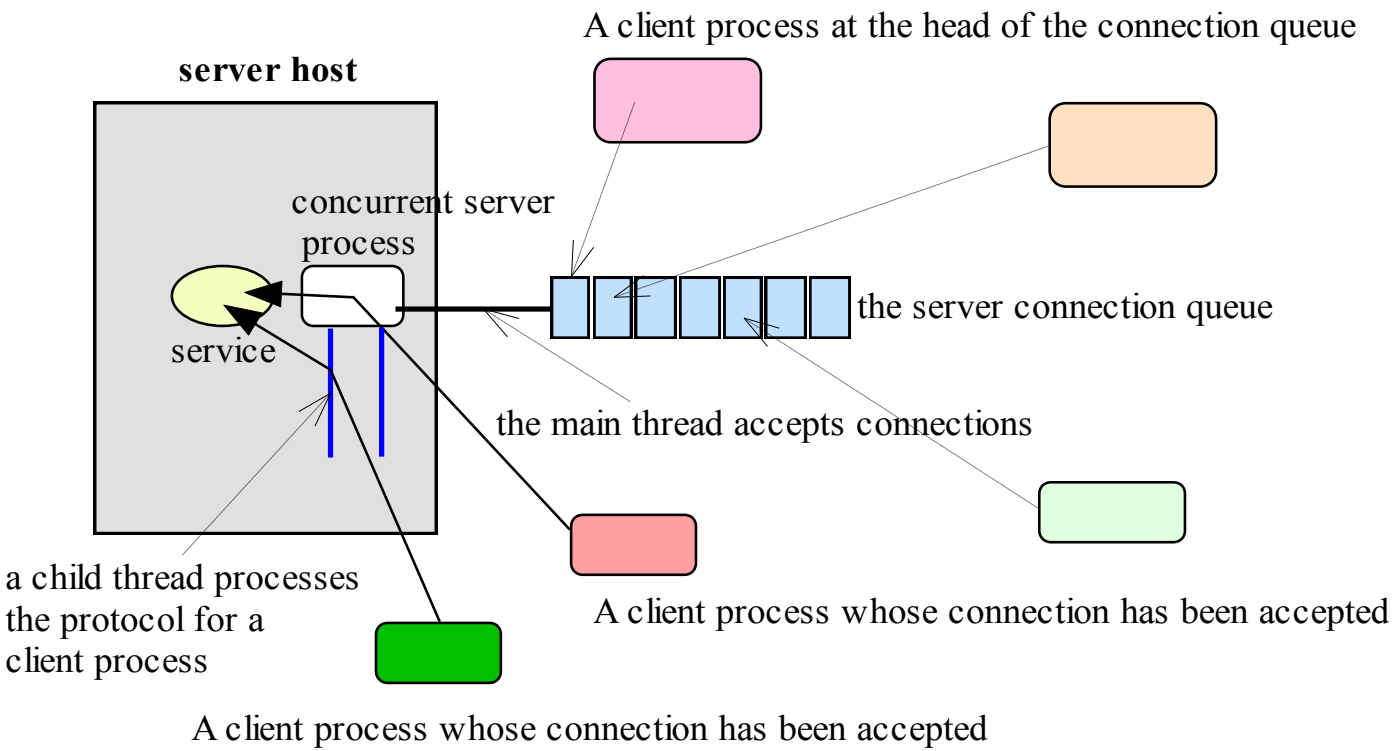


Diagrama de seqüencia de Servidor orientat a la connexió amb Threads

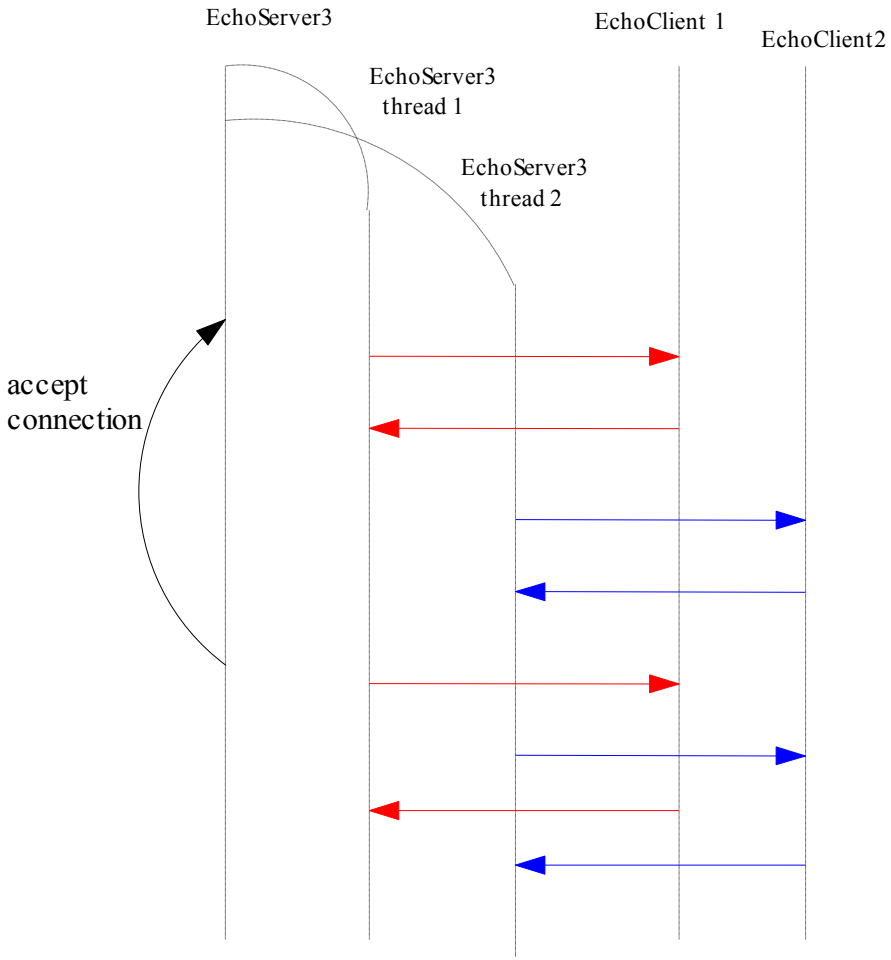
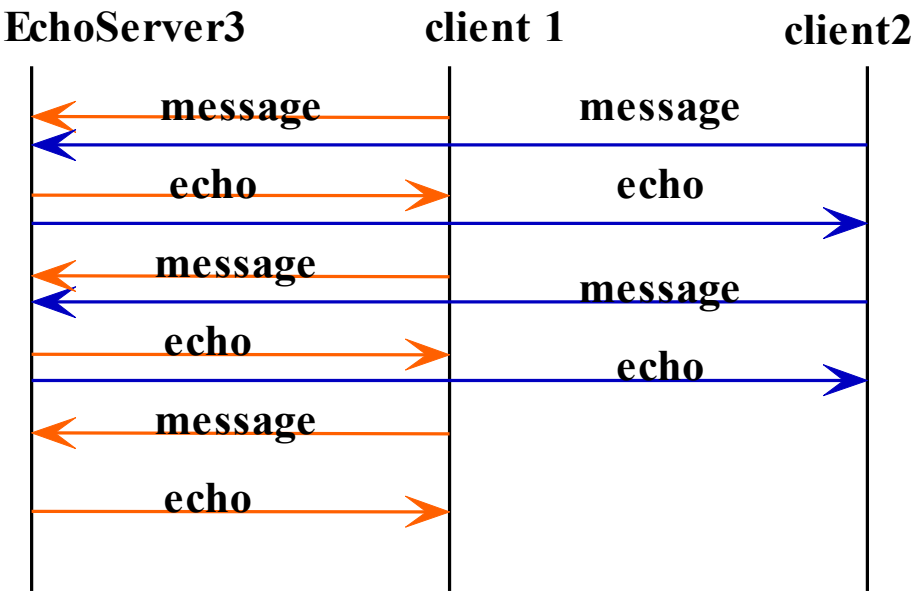


Diagrama d'esdeveniments de Servidor orientat a la connexió amb Threads



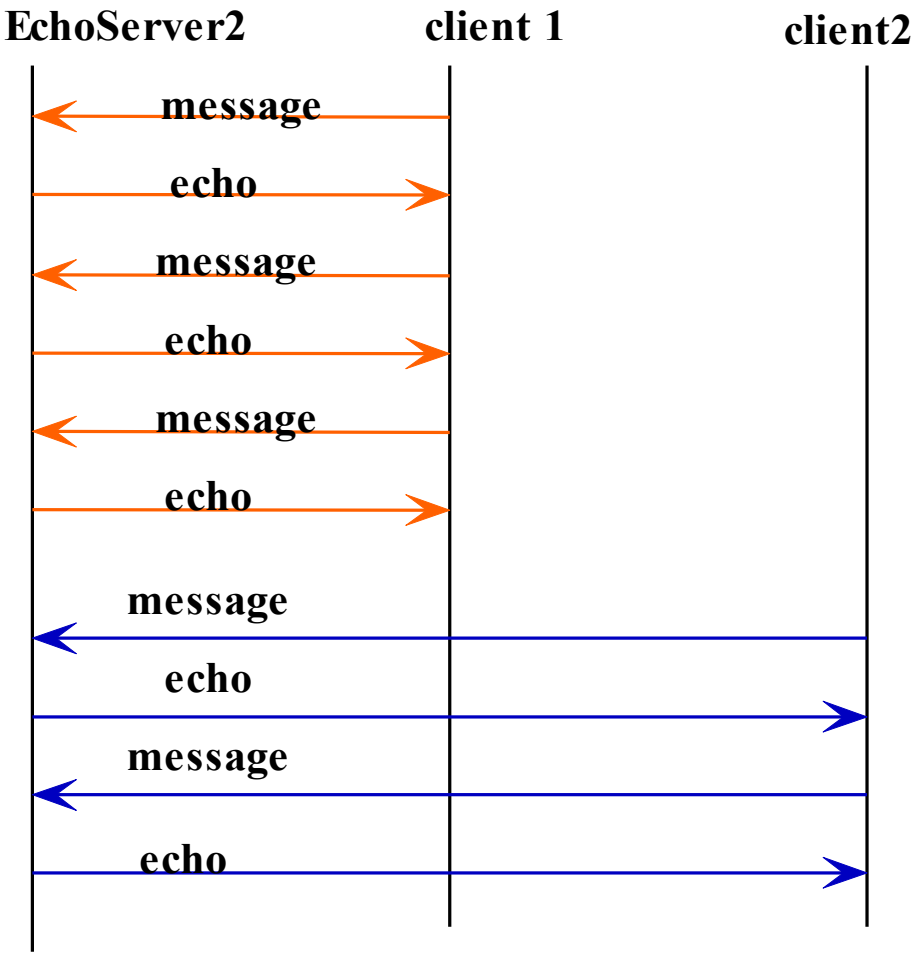
Threads amb JAVA

Threads amb JAVA

Exemple Servidor orientat a la connexió amb Threads

Source Code DateServer with Threads

Diagrama d'esdeveniments de Servidor orientat a la connexió iteratiu seqüencial



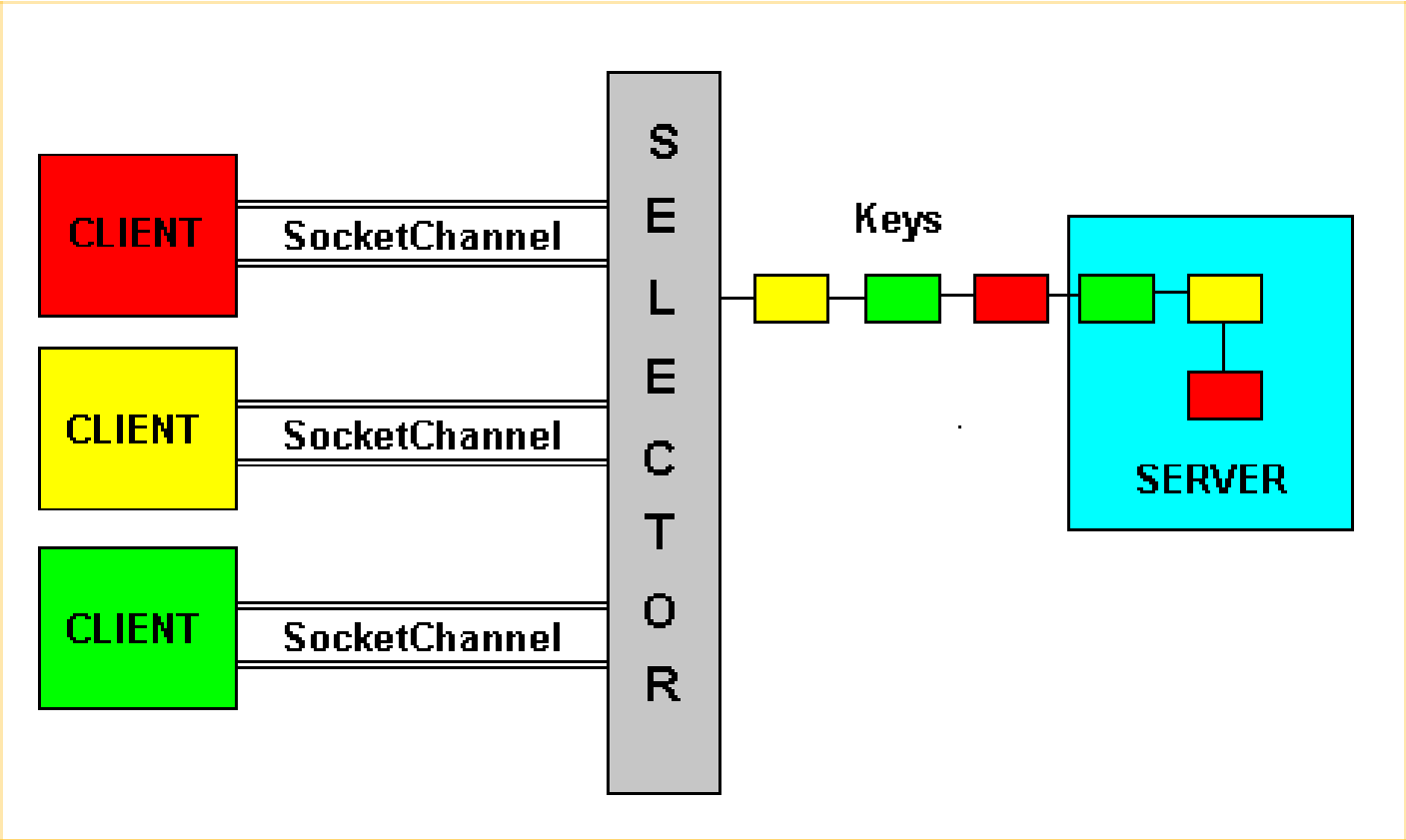
Exemple Servidor orientat a la connexió seqüencial

Source Code Sequential Server Example

Servidor orientat a la connexió iteratiu fent servir Selector

- **Problema:** Gestió de les operacions bloquejants (accept, read)
- *Solució:* Nonblocking InputOutput (nio)
- Link API java.nio

Selector


CLIENT
SocketChannel
CLIENT
SocketChannel
CLIENT
SocketChannel
SELECTOR
Keys
SERVER

Servidor orientat a la connexió iteratiu fent servir Selector

Crear un *ChannelSocket* no bloquejant:

```
ServerSocketChannel server = ServerSocketChannel.open();
server.socket().bind(new java.net.InetSocketAddress(8000)
server.configureBlocking(false);
```

Fer servir *Selector* per gestionar els canals:

```
Selector selector = Selector.open();
SelectionKey serverkey = server.register(selector,
SelectionKey.OP_ACCEPT);
```

Exemple Servidor no Bloquejant fent servir Selector

- Source Code: ServerSelector
- ServerSelector GIST

Consideracions d'implementació

- **Gestió d'excepcions:**
  - Totes les excepcions s'han de tractar de forma adequada amb un `try catch`.
  - Cada tipus d'excepció ha de tenir el seu propi `catch`.
  - El `finally` després d'un try catch s'executa **sempre**, hi hagi o no excepció.
  - Usar `finally` per tancar els sockets sempre un cop acabada la connexió.
- **Tipus d'excepcions d'IO**
  - Broken pipe, No s'ha pogut escriure perquè s'ha desconnectat l'altra part. (`IOException`)
  - Socket propi tancat o no es pot crear, no es pot accedir al socket (`SocketException`)
  - Socket timeout, s'ha excedit el temps d'espera en el socket propi (`InterruptedIOException`)
  - Protocol Excepetion Problema en la capa de TCP (`ProtocolException`)

UNIVERSITAT DE BARCELONA

Consideracions d'implementació

Consulteu l'API de JAVA de sockets per saber cada funció quina excepció llança

Socket API

Consideracions d'implementació

- **Gestió TimeOuts:**
  - Tot Socket de dades que realitzi operacions de read ha de tenir un TimeOut adequat.
  - El temps del timeOut no ha de ser gaire elevat. Preten trobar fallades en la connexió.
  - Si fem una aplicació que pot tenir elevat temps d'espera, llavors un cop s'activa el timeOut cal comprovar si l'stream d'entrada està disponible (`available`), en cas afirmatiu es torna a fer l'operació de lectura.
- **Gestió Threads:**
  - Els threads s'han d'acabar de forma natural, un cop acabada la seva tasca.
  - Abans d'acabar-se el thread, ha de tancar tots els sockets que tingués oberts.

Comunicació unicast vs multicast

Unicast: Enviament cap a un únic receptor

- TCP: establint connexió
- UDP: sense establir connexió, no fiable

Multicast: Enviament cap a múltiples receptors.

- UDP: sense establir connexió, no fiable

Comunicació multicast mitjançant datagrames

IP multicast

- Construït a sobre del Internet Protocol (IP),
- Només disponible via UDP! No hi ha garanties de recepció.
- Permet enviar un paquet IP a un conjunt de màquines que formin un grup multicast.

Grup multicast

- Identificat amb una adreça d'internet de la Classe D: 224.0.0.0 fins 239.255.255.255.
- Quant una màquina s'afegeix a un grup multicast, rep tots els missatges que s'hi enviïn.
- Es poden enviar paquets sense formar-ne part.

Exemple IP multicast amb java

Exemple: `java MulticastPeer Hola 224.0.0.1`

** La IP 224.0.0.1 es refereix al grup multicast de la xarxa local.*

Source Code MulticastPeer