



# Pràctica 2: Sockets

## Aplicacions i serveis d'internet — iTIC

Francisco del Àguila López      Aleix Llusà Serra      Alexis López Riera

2 de març de 2018

### Índex

<b>1</b>	<b>Organització</b>	<b>1</b>
1.1	Objectius . . . . .	1
1.2	Condicions . . . . .	2
1.3	Lliurables . . . . .	2
1.4	Material necessari . . . . .	2
<b>2</b>	<b>Requeriments de la pràctica</b>	<b>2</b>
<b>3</b>	<b>Introducció als Sockets</b>	<b>3</b>
3.1	Arquitectura Client / Servidor . . . . .	3
<b>4</b>	<b>Exemples de Codi</b>	<b>4</b>
4.1	Exemples de shell . . . . .	4
4.2	Exemples amb Python . . . . .	5
4.3	Exemples amb C . . . . .	6
<b>5</b>	<b>Pipes</b>	<b>6</b>
<b>6</b>	<b>Utilitat Select()</b>	<b>7</b>
6.1	Exemples amb <code>select()</code> . . . . .	7

### Resum

Comunicació entre processos a través de xarxa: Sockets

## 1 Organització

### 1.1 Objectius

Els objectius d'aquesta pràctica són:

1. Entendre la comunicació entre processos basada en sockets.
2. Permetre la comunicació entre diferents màquines connectades en xarxa.
3. Entendre el concepte de servidor i de client.

4. Observar com la comunicació per sockets és independent del llenguatge de programació.
5. Ser capaç de dissenyar aplicacions en xarxa.

## 1.2 Condicions

- La pràctica està calibrada per a ésser treballada en equips de dues persones.
- La durada de la pràctica és de 1 setmana.

## 1.3 Lliurables

Heu d'entregar:

- El codi font de les tasques que se us demanen.

## 1.4 Material necessari

Per tal de fer servir els sockets de diferents maneres cal disposar de:

- Eines de compilació de C.
- Intèrpret de Python.
- L'aplicació `netcat`.

[TASCA PRÈVIA 1](#) Instal·leu el que us faci falta per realitzar aquesta pràctica. Instal·leu el paquet `netcat`.

## 2 Requeriments de la pràctica

L'objectiu d'aquesta pràctica és poder fer un xat 1 a 1 entre dues màquines. Per aconseguir-ho, una de les màquines ha de ser servidora (espera una connexió) i l'altre ha de ser client (es connecta a un servidor). Per tant, s'ha de dissenyar una única aplicació que en el moment d'executar-la s'ha de ficar bé en mode client o en mode servidor (aquest mode pot ser escollit en funció dels paràmetres que se li passen). Si es vol canviar de mode, s'ha de tornar a executar l'aplicació.

Si l'aplicació està en mode servidor, només ha d'acceptar una única connexió de client. Per poder acceptar més d'una connexió de client, hi ha dues possibilitats:

- Crear tants processos fills com connexions de client es tinguin.
- Gestionar les diferents connexions com si fossin diferents fitxers des de on es poden llegir/escriure dades. Per poder fer això sense conèixer per quina connexió vindrà la nova dada, s'ha d'aprofitar una de les eines disponibles del S.O. anomenada `select` que es presenta més endavant.

### 3 Introducció als Sockets

Els *Sockets* és el mecanisme que permet la comunicació entre processos proporcionada pel sistema operatiu. Aquests processos poden estar presents a la pròpia màquina o bé a altres màquines.

Existeixen essencialment dos tipus de *Sockets* en funció del seu domini de comunicació. El domini de comunicació ens indica on es troben els processos que s'intercomunicaran.

- Si els processos es troben en la mateixa màquina, el domini de comunicació queda definit per `AF_UNIX`.
- Si els processos es troben a distintes màquines connectades amb una xarxa TCP/IP, el domini de comunicació serà `AF_INET`.

A dins del domini de comunicació `AF_INET` es pot subdividir els tipus de *Sockets* tenint en compte el tipus de protocol que faran servir.

- *Sockets Stream* si fan servir el protocol TCP per intercanviar els missatges entre els processos.
- *Sockets Datagram* si fan servir el protocol UDP per intercanviar els missatges entre els processos.
- *Sockets raw* no fan servir ni TCP ni UDP, poden servir per definir nous protocols o aplicacions sense utilitzar la capa de transport.

Les diferències principals entre els *Sockets Stream* i els *Sockets Datagram* estan associades al fet de fer servir TCP o UDP, per tant en els *Sockets Stream* cal que els processos que volen intercanviar missatges estableixin una connexió prèvia. El protocol TCP garanteix que els missatges arribaran correctament i pot provocar que els processos quedin bloquejats esperant o intercanviant missatges fins que la connexió s'alliberi. Per contra, en els *Sockets Datagram* no cal cap tipus de connexió prèvia i per tant no queda garantit que l'altra procés estigui escoltant o no. En aquest cas no es provocarà que el procés quedi bloquejat en l'intercanvi de dades.

#### 3.1 Arquitectura Client / Servidor

El mecanisme que possibilita que dos processos es comuniquin, generalment està basat en el model de Client / Servidor. En aquest model el client és qui es connecta cap al servidor, per tant és qui pren la iniciativa de la comunicació i necessita conèixer quina identificació té el servidor, que en Internet és la adreça IP. Per contra, el servidor és qui espera rebre peticions per part dels clients, per tant no necessita conèixer cap tipus d'identificació de qui es connectarà.

Els processos que proporcionen serveis d'aplicació són servidors, per tant creen sockets en el moment d'arrencar que estan en l'estat de *listening*. Aquests sockets estan esperant les connexions dels programes clients.

Un servidor TCP (*Sockets Datagram*) pot donar servei a varis clients concurrentment per mitjà de la creació de processos fills per cadascuna de les connexions TCP establertes de cada client. Es crea un socket per cada connexió. Aquestes connexions estan en mode *established*, per tant, hi ha una connexió virtual de socket a socket coneguda com a sessió TCP. Mentre la connexió està establerta hi ha un intercanvi de bytes en mode duplex.

Un servidor pot crear varies connexions TCP amb el mateix port local i adreça local ja que cada client connectat té el seu propi procés fill, i per tant el seu propi socket. Aquestes connexions són tractades com a sockets diferents pel sistema operatiu ja que el socket queda definit pel parell ip/port de les adreces origen i destí conjuntament.

En el cas s'un servidor UDP (*Sockets Datagram*) no hi ha connexió establerta. Per tant no es pot determinar si hi ha comunicació amb diferents clients. Un servidor UDP no crea un nou procés fill per cada client. El mateix procés servidor gestiona tots els paquets que li arriben de diferents clients de manera seqüencial a través del mateix socket. Per tant el socket només queda definit pel parell ip/port local i no pel ip/port remot. De totes maneres, per cada paquet que arriba, que està associat al mateix socket del servidor, és possible que el procés servidor conegui quina és l'adreça remota que ha enviat aquest missatge, i per tant pugui generar una resposta particularitzada al procés client.

Els passos que intervenen en una connexió de client són:

1. Crear un socket amb la crida de sistema *socket()*
2. Connectar el socket a l'adreça del servidor amb la crida *connect()*
3. Enviar i rebre dades fent servir per exemple les crides *read()* i *write()*

En el cas del socket de servidor, els passos són:

1. Crear un socket amb la crida de sistema *socket()*
2. Unir el socket a una adreça i port local de la màquina servidora amb la crida de sistema *bind()*
3. Escoltar possibles connexions amb la crida *listen()*
4. Acceptar la connexió amb la crida *accept()*. Aquesta crida és bloquejant fins que el client connecti amb el servidor.
5. Enviar i rebre dades

## 4 Exemples de Codi

### 4.1 Exemples de shell

Una eina important per treballar amb *Sockets* des de la consola és **netcat** o també simplificat **nc**. Amb aquesta eina entre altres coses es poden establir connexions **TCP** o intercanviar dades **UDP**. La mateixa eina pot actuar tant de client com de servidor. Aquesta eina permet que el que s'escriu per teclat s'envia pel socket establert i el que es rep pel socket s'escriu a pantalla.

**TASCA 2** Executeu el **netcat** per acceptar connexions **TCP** pel port 10000. Executeu el **netcat** per connectar-se amb una altra màquina pel port 10000. Intercanvieu missatges entre elles. Podríeu des de una màquina client executar comandes en la màquina que escolta?

**TASCA 3** Executeu el **netcat** per acceptar dades **UDP** pel port 10000. Executeu el **netcat** per enviar dades UDP a una altra màquina pel port 10000. Intercanvieu missatges entre elles.

En el cas que es connectin dos clients al mateix servidor, el servidor ha d'estar pendent de cada client. En el cas del **TCP** s'han d'establir dos connexions, però en el cas de **UDP** no existeixen connexions.

TASCA 4 Si un sol procés és el que fa de servidor, tant en el cas de **TCP** com en el de **UDP**, es reben els missatges simultàniament dels dos clients?

TASCA 5 Si es fa servir **netcat** com aplicació servidora, en quins casos es poden connectar els dos clients? Comproveu els resultats de forma pràctica i doneu una explicació.

Observeu que el comportament que té **netcat** per **UDP** no correspon amb el que hauria de tenir.

## 4.2 Exemples amb Python

Per treballar amb *Sockets* i Python es pot fer servir la llibreria de *Sockets* <http://docs.python.org/2/library/socket.html>. En aquesta llibreria està definit tot el que cal per treballar amb els *Sockets*. Feu una llegida per saber què és cada apartat. Al final es troba un exemple de *Sockets* client i servidor basat en **TCP**.

TASCA 6 Comproveu el funcionament d'aquests exemples simples.

Heu de comprovar el següent:

- L'exemple de client es limita a obrir el socket, enviar dades, comprovar la possible resposta i finalment tancar el socket. Per tant no deixa la possibilitat de tornar a enviar noves dades.
- L'exemple de servidor es limita a esperar a que es produeixi una connexió, dir qui s'ha connectat, rebre dades del client i tornar-les a enviar. Queda en un bucle mentre la connexió està establerta, però un cop s'ha tancat s'acaba el procés i no torna a esperar noves connexions.
- Tant el client com el servidor són un únic procés, això vol dir que, especialment en el cas del servidor, quan s'ha establert una connexió, el procés servidor deixa d'escollar, passa a atendre la connexió i per tant no aten més possibles connexions.

En el cas de les aplicacions servidores importants es creen processos fills per atendre les connexions mentre que el procés pare pot continuar esperant noves possibles connexions. Això fa que el disseny d'una aplicació servidora sigui més complex.

TASCA 7 Comproveu que es poden comunicar perfectament els exemples de Python fets en **TCP** amb els exemples de **netcat**.

TASCA 8 Transformeu aquests exemples per fer servir una comunicació **UDP**. Recordeu que en el cas del **UDP** no existeixen les connexions. Un mètode útil en aquest cas és **recvfrom** per poder distingir qui envia les dades, ja que varis clients poden enviar dades al mateix servidor.

TASCA 9 Comproveu si en el cas **UDP** es poden comunicar dos clients amb un sol servidor.

TASCA 10 Dissenyeu una aplicació d'un únic procés en Python que permeti fer un xat entre dues màquines en **TCP** i només accepti una sola connexió. S'ha de permetre que tant client com servidor puguin anar enviant dades en qualsevol moment sense tenir l'obligació que l'altre interlocutor hagi de respondre.

TASCA 11 *Opcional* Milloreu l'aplicació anterior permetent múltiples clients simultanis.

Per aquestes dues últimes tasques pot ser útil aprofitar la eina `select()` explicada en l'últim apartat.

TASCA 12 *Opcional* Milloreu l'aplicació anterior permetent múltiples processos. Es poden generar tants processos fills com connexions o entrades s'han d'atendre.

### 4.3 Exemples amb C

La programació de *Sockets* amb C és molt similar al cas de Python. Com ja sabeu, les estructures de dades amb C són més primitives i per tant no són tan còmodes de fer servir com en Python. Però per fer servir els sockets amb C s'ha de seguir la mateixa estructura general que en Python. En el fons, els sockets són una funcionalitat del S.O. i no de C ni Python. Tant C com Python com la resta de llenguatges de programació tan sols els presenten d'una manera amigable.

A [http://www.linuxhowtos.org/C\\_C++/socket.htm](http://www.linuxhowtos.org/C_C++/socket.htm) hi ha un parell d'exemples client / servidor de com s'han de fer servir els sockets. Llegiu el codi amb detall entenent que és el que es fa i llegint les explicacions associades en aquesta web.

TASCA 13 Comproveu el funcionament dels exemples proposats. Combineu les aplicacions client / servidor amb les aplicacions fetes amb Python i amb el `netcat`.

Recordeu que sempre teniu disponible el manual de sockets de Linux *man socket*.

TASCA 14 Modifiqueu els exemples per aconseguir l'aplicació de xat proposada. Comproveu que aquestes aplicacions són compatibles amb les aplicacions desenvolupades amb Python.

TASCA 15 *Opcional* Modifiqueu l'aplicació anterior de manera que aprofiti la funció `select()` del sistema operatiu.

## 5 Pipes

Amb Linux es poden encadenar les entrades i sortides d'aplicacions a altres aplicacions. Per exemple, la sortida de `ls` la podem passar a un `grep` per buscar una paraula:

```
ls | grep patró
```

Això ho podem fer també amb els *sockets* que s'han fet servir anteriorment. Per exemple, fent servir l'aplicació `netcat` es pot enviar des de un client amb `netcat` un text qualsevol a un servidor i que aquest servidor filtri la paraula patró qualsevol. Per fer això, el servidor de `netcat` s'hauria de connectar a un `grep` així:

```
nc -l 5000 | grep patró
```

Si volem que la sortida d'aquest `grep` es retorni cap al client. Un mecanisme simple de muntar és un segon *socket* amb un altre `netcat` que envii el resultat cap el client. Però aquesta solució no és gens elegant ja que tenim 2 *sockets* que s'estan fent servir de manera unidireccional. Com heu pogut comprovar els *sockets*, ja siguin client o servidor, permeten una comunicació bidireccional.

Per aconseguir que amb un mateix *socket* tant s'envii la informació que es vol processar, en aquest cas per un *grep*, com es rebi la resposta d'aquest procés, cal la existència auxiliar d'una *pipe* bidireccional. Aquesta *pipe* bidireccional s'aconsegueix amb la comanda *mkfifo*. Aquesta *pipe* bidireccional és molt similar a un fitxer, apareixent en el sistema de fitxers com a tal. Un cop creada aquesta *pipe* el que s'ha de fer és:

```
nc -l 5000 < fifo | grep --line-buffered patró > fifo
```

On *fifo* és la *pipe* bidireccional creada amb *mkfifo*. Es pot observar que l'entrada a *nc* serà el contingut de la *fifo*, la sortida del *nc* és l'entrada del *grep* i la sortida del *grep* va a parar a la *fifo*. Per tant, el que va a parar a la *fifo* s'enviarà pel *socket* i per tant arribarà al client com resposta, i el que va a parar al *socket* (correponent al que envia el client) va a parar al *grep*. L'opció *--line-buffered* del *grep* cal per a que pogui donar una sortida a cada línia i no s'espera al final del fitxer.

## 6 Utilitat `Select()`

En moltes ocasions cal llegir diferents entrades com són les del teclat, el port serie, etc. Per poder llegir aquestes entrades es disposa de funcions que ho fan. Generalment aquestes funcions són bloquejants, això vol dir que el programa queda parat en aquest punt fins que hagi dades disponibles. Si aquest programa vol fer lectures de més d'una entrada però no està establert en quina entrada apareixen les dades, s'ha de determinar de quina entrada s'ha de llegir primer.

Una alternativa per abordar aquest problema és generar tants processos fills com entrades es vulguin llegir. Això implica que s'ha d'establir un mecanisme de comunicació entre els diferents processos. Per altra banda, a priori, l'execució de diferents processos provoca una demanda de recursos de sistema superior a l'execució d'un sol procés.

Una altra alternativa més simple i que permet solucionar el problema amb un sol procés és aprofitar una de les eines que ofereix el sistema operatiu per gestionar múltiples entrades. Aquesta eina en els sistemes basats en UNIX és el `select()`.

### 6.1 Exemples amb `select()`

El `select()` és una utilitat que ofereix el sistema operatiu, per aquest motiu està disponible independent del llenguatge de programació. En tot cas, el llenguatge de programació utilitzat la oferirà amb un marc adequat per fer més simple la seva utilització.

A continuació es presenta un exemple d'ús del `select()` en l'entorn de Python, aprofitant les característiques d'alt nivell d'aquest llenguatge. Una descripció exhaustiva del `select()` en Python es troba a <https://docs.python.org/3/library/select.html>.

En aquest exemple es vol fer una lectura de les entrades de teclat i de ratolí. L'entrada de teclat és l'estàndard input, que és equivalent a considerar el fitxer `sys.stdin`. Per altra banda, aprofitant que en els sistemes UNIX qualsevol cosa és un fitxer, l'entrada de ratolí es pot llegir a partir del fitxer de dispositiu `/dev/input/mice`, al qual l'usuari `root` o bé el grup `input` pot accedir.

```
import sys
import select
```

```

import struct

def wait_input(llista):
    rlist, _, _ = select.select(llista, [], [])
    if rlist[0] == llista[0]:
        data = sys.stdin.readline()
        print "Teclat:_" + data
    elif rlist[0] == llista[1]:
        pos = getMouseEvent()
        print ("L:%d, M: %d, R: %d, x: %d, y: %d" % pos );
    else :
        print "Impossible"

def getMouseEvent():
    buf = file.read(3);
    button = ord( buf[0] );
    bLeft = button & 0x1;
    bMiddle = ( button & 0x4 ) > 0;
    bRight = ( button & 0x2 ) > 0;
    x,y = struct.unpack( "bb", buf[1:] );
    datam = ( bLeft, bMiddle, bRight, x, y )
    return datam

file = open( "/dev/input/mice", "rb" );

file_d=[sys.stdin, file]

while 1:
    wait_input( file_d )
file.close()

```

La funció `getMouseEvent()` simplement dona format a les dades recollides del ratolí. La variable `file_d` és una llista amb els identificadors dels fitxers que intervenen. A la funció `wait_input(llista)` és on s'utilitza el `select()`. En aquest exemple se li passa com a primer paràmetre una llista amb els fitxers amb els que s'espera entrada, els altres paràmetres no es fan servir. Com a sortida retorna en al primer paràmetre la llista dels fitxers que han tingut alguna entrada. La resta de la funció s'encarrega de fer les accions oportunes en cada cas.