

Desenvolupament d'aplicacions de comunicacions: Sockets

Cèsar Fernández, Enric Guitart, Carles Mateu

Departament d'Informàtica
Universitat de Lleida

Febrer 2017



Esquema

1 Conceptes generals

- Introducció
- Model igual-a-igual
- Model client-servidor
- Berkeley Socket API

2 Berkeley Sockets

- Estructures de dades
- Adreces i noms
- Nocions generals
- Especificació de les crides
- Estructures client/servidor

3 Comunicacions en Python

- Mòdul socket
- Mòdul select
- Mòdul asyncore

4 Protocols

- Introducció
- Diagrama de temps - PDU



Introducció

A l'hora de comunicar entitats d'una xarxa podem emprar dos models o arquitectures:

- Model igual-a-igual (*peer-2-peer*)
 - Patró d'interacció entre aplicacions corporatives descentralitzades
- Model client/servidor
 - Patró d'interacció entre aplicacions corporatives centralitzades



Model igual-a-igual

Peer

Qualsevol programa que ofereix un servei que pot ser accedit des de la xarxa i que al seu torn accedexi a serveis proporcionats per altres

Exemple

GNutella: transferència de fitxers

- Els programes escolten un port i responen peticions de fitxers que els arriben
- Els programes envien peticions de fitxers als seus iguals al mateix port



Model client-servidor

Servidor

Qualsevol programa que ofereix un servei que pot ser accedit des de la xarxa

Client

Qualsevol programa que tramet una demanda al servidor i espera la seva resposta

Exemple

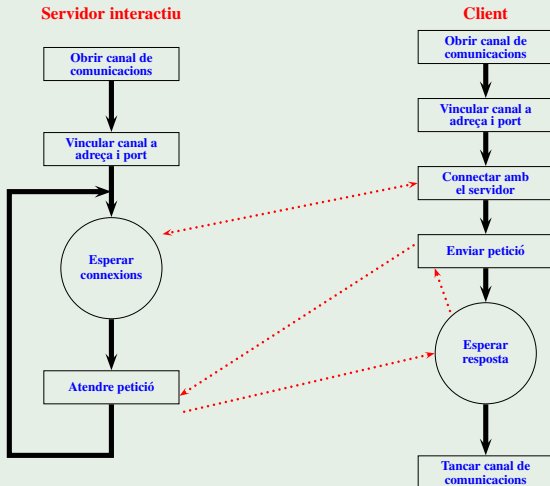
Servei d'eco UDP:

- El servidor escolta pel port 7 i repeteix qualsevol paquet que entri per aquest port
- El client envia un paquet a un servidor determinat, esperant la seva resposta
- Mecanisme emprat per l'aplicació `ping`



Model client-servidor

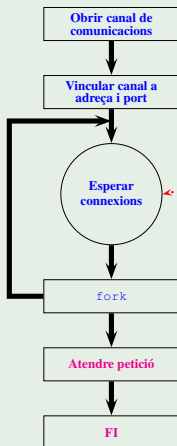
Estructura client/servidor interactiu



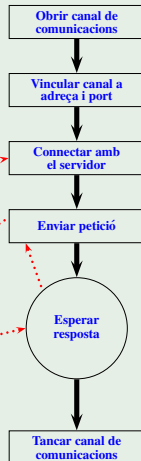
Model client-servidor

Estructura client/servidor concurrent

Servidor concurrent



Client



Berkeley Socket API

- Interfície de programació per comunicació entre programes mitjançant descriptors de fitxers
- *BSD Sockets*: cas particular per protocol IP. Altres protocols suportats (Unix-Domain Sockets, X25 Sockets, ATM PVC/SVC, Appletalk, IPv6, etc.)
- Mantenen la característica asimètrica del model client-servidor
- Existeixen altres API: TLI UNIX-SV, JAVA, etc.
- Coherent amb la filosofia UNIX: empra crides de tractament d'arxius (`open`, `close`, `read`, ...)
- Permet establir serveis orientat o no a connexió emprant TCP o UDP



Tipus de sockets

- *Stream Sockets*: Orientats a connexió, protocol TCP
 - Comunicació bidireccional fiable: manteniment de l'ordre, lliure d'errors, lliurament assegurat.
 - Ex: HTTP (Web), SMTP/POP3 (e-mail).
- *Datagram Sockets*: No orientats a connexió, protocol UDP
 - Comunicació no fiable: no s'assegura ni el lliurament ni l'ordre.
 - Ex: NFS (Compartició de disc), DNS (Resolució de noms).
- *Raw Sockets*: Construcció manual dels paquets.



Estructures de dades

- Definides a `/usr/include/sys/socket.h` del tipus:

```
/usr/include/sys/socket.h
```

```
struct sockaddr {  
    u_short    sa_family; /*Familia adreces*/  
    char       sa_data[14]; /*Fins 14 octets específics*/  
};
```

- `sa_family`: Especifica la família de protocols
 - `AF_INET` → TCP/IPv4
 - `AF_UNIX` → Unix
 - `AF_INET6` → TCP/IPv6
 - `AF_IPX` → IPX
 - ...
- `sa_data`: Varia en funció del protocol



Estructures de dades

- L'estructura corresponent a `AF_INET`, es troba a:
`/usr/include/netinet/in.h` .

```
/usr/include/netinet/in.h
```

```
struct sockaddr_in {  
    short sin_family;          /*AF_INET*/  
    u_short  sin_port;         /*16 bits pel port*/  
    struct    in_addr sin_addr; /*32 bits adreca IP*/  
    char      sin_zero[8];     /*no emprat*/  
};
```

```
struct in_addr {  
    unsigned long s_addr;  
};
```

- `sin_port` i `sin_addr` han d'estar en *network byte order*



Estructures de dades: L'ordenació dels octets

- Diferents architectures ordenen els octets de les paraules de 16 i 32 bits de forma diferent
- Necessitat de transmetre-les de forma estàndard
- Funcions de traducció (definides a `#include <netinet/in.h>`)
 - `htons()` Host a xarxa per enters curts (16 bits)
 - `htonl()` Host a xarxa per enters llargs (32 bits)
 - `ntohs()` Xarxa a host per enters curts (16 bits)
 - `ntohl()` Xarxa a host per enters llargs (32 bits)



Estructures de dades: Les adreces IP (I)

- El API *Berkeley Sockets* proporciona una serie de funcions que ens permeten convertir adreces a `struct sockaddr_in`.
 - `inet_addr()` Adreça en format text a `sockaddr_in` (format xarxa).
 - `inet_ntoa()` Adreça en format `sockaddr_in` a text.
- `inet_addr()` ja retorna l'adreça en format de xarxa i no cal cridar a `hton()`.



Estructures de dades: Les adreces IP (II)

- El API *Berkeley Sockets* també proporciona mitjans per convertir de noms de màquina a adreça IP.
- Per fer-ho emprà l'estructura:

```
/usr/include/netdb.h  
  
#include <netdb.h>  
struct hostent{  
    char    *h_name;  
    char    **h_aliases;  
    int      h_addrtype;  
    int      h_length;  
    char    **h_add_list;  
};
```



Estructures de dades: Les adreces IP (III)

- L'estructura `hostent` es usada per la funció `gethostbyname` per retornar la informació d'adreces corresponent a un nom concret.

gethostbyname

```
struct hostent *gethostbyname(const char *name);
```

- Aquest crida cerca, segons la definició a `/etc/resolv.conf` quina informació d'adreces i sobrenoms té el nom que li passem.

Exemple

```
[root@dire pl]# ./info_host www.linux.com
Resultat:
Nom Oficial:      www.linux.com
Alias:
Tipus d'adreca:  AF_INET -> IPv4
Longitud d'adreca: 4 bytes
Adreces:
- 140.211.167.51
- 140.211.167.50
```

Crides de BSD sockets

Crida	Descripció
<code>socket()</code>	Crear socket
<code>bind()</code>	Fixar adreça
<code>listen()</code>	Especificar cua
<code>accept()</code>	Esperar connexió
<code>read()</code> , <code>write()</code> <code>recv()</code> , <code>send()</code> <code>recvfrom()</code> , <code>sendto()</code>	Transferir dades
<code>close()</code>	Tancar socket

- Els arxius de definició són `sys/types.h` i `sys/socket.h`
- La implementació depèn de si es tracta de:
 - Servidor o client
 - Orientat a connexió o datagrama



Crides de BSD sockets: `socket`

- Crea un socket del tipus especificat i ens retorna un descriptor de socket (un enter) que emprarem per la resta de crides, o en cas d'error a la creació del socket ens retorna -1.

socket

```
int socket(int família, int tipus, int protocol);
```

- *família* = `AF_INET` per als protocols TCP/IP, `AF_UNIX` per Unix sockets, etc.
- *tipus* = (`SOCK_STREAM`, `SOCK_DGRAM`) per als protocols TCP, UDP i (`SOCK_RAW`) *raw sockets* respectivament
- *protocol* = 0 per acceptar les combinacions per defecte (o bé `IPPROTO_TCP`, `IPPROTO_UDP`, etc.)



Crides de BSD sockets: `bind`

- “Enganxa” un socket a un port (i adreça) determinats per on rebrem les connexions. Retorna -1 en cas d’error.

`bind`

```
int bind(int sockfd, struct sockaddr *adreça, int longitud) ;
```

- *sockfd* = descriptor de socket (de `socket()`)
- *adreça* = adreça i port on “enganxarem” el socket
- *longitud* = mida (en octets) de: *adreça*
- Podem escollir un port aleatori i deixar que el sistema operatiu ens posin l’adreça de la forma:

```
adrec.a.sin_port = 0; /* Port aleatori */  
adrec.a.sin_addr.s_addr= INADDR_ANY; /* Adr. autom. */
```

Crides de BSD sockets: `connect`

- Estableix la connexió cap a un servidor a l'adreça i ports especificats. Retorna 0 si s'ha pogut connectar, -1 en cas d'error.

`connect`

```
int connect(int sockfd, struct sockaddr *adreça_serv,  
int longitud);
```

- *sockfd* = descriptor de socket (de `socket()`)
- *adreça_serv* = adreça i port on volem connectar
- *longitud* = mida (en octets) de: *adreça_serv*
- No es necessari cridar a `bind` per connectar, al cridar a `connect` ell crida a `bind` si fa falta.



Crides de BSD sockets: `listen`

- Posa el socket en espera de connexions. Retorna 0 si s'ha pogut realitzar, -1 en cas d'error.

`listen`

```
int listen(int sockfd, int #_connex);
```

- *sockfd* = descriptor de socket (de `socket()`)
- *#_connex* = nombre màxim de connexions en cua



Crides de BSD sockets: `accept`

- Accepta una de les connexions que tenim en la cua d'espera. Retorna un nou socket per la transferència de dades o -1 en cas d'error.

`accept`

```
int accept(int sockfd, struct sockaddr *adreça_remota,  
int longitud) ;
```

- *sockfd* = descriptor de socket (de `socket()`)
- *adreça_remota* = identificació de la màquina remota (adreça i port)
- *longitud* = mida (en octets) de: *adreça_remota*
- Al cridar a `accept()` una de les connexions que tinguem pendents a la cua establerta per `listen()` es acceptada, si no n'hi ha s'espera a que n'hi hagi.



Crides de BSD sockets: `send`

- Envia dades per un socket de tipus *connectat*(STREAM). Retorna el nombre de octets enviats (pot ser menys del que li passem) o -1 en cas d'error.

`send`

```
int send(int sockfd, char *buff, int #octets, int flags) ;
```

- *sockfd* = descriptor de socket (de `socket()`)
- *buff* = dades a enviar
- *#octets* = longitud a enviar
- *flags* = paràmetres especials, generalment 0 (MSG_OOB, MSG_DONTROUTE, etc.)



Crides de BSD sockets: `recv`

- Recull dades rebudes per un socket de tipus *connectat*(STREAM). Retorna el nombre de octets rebuts (no pot superar el màxim que li donem nosaltres) o -1 en cas d'error.

`recv`

```
int recv(int sockfd, char *buff, int #_octets, int flags) ;
```

- *sockfd* = descriptor de socket (de `socket()`)
- *buff* = dades a rebre
- *#_octets* = longitud màxima a rebre
- *flags* = paràmetres especials, generalment 0 (MSG_OOB, MSG_DONTROUTE, etc.)



Crides de BSD sockets: `sendto`

- Envia dades per un socket de tipus *no-connectat*(DGRAM). Retorna el nombre de octets enviats (pot ser menys que *#_octets*) o -1 en cas d'error.

`sendto`

```
int sendto(int sockfd, char *buff, int #_octets, int flags, struct sockaddr *cap_a, int longitud);
```

- *sockfd* = descriptor de socket (de `socket()`)
- *buff* = dades a enviar
- *#_octets* = longitud a enviar
- *flags* = paràmetres especials, generalment 0 (MSG_OOB, MSG_DONTROUTE, etc.)
- *cap_a* = adreça destinació
- *longitud* = longitud en octets de *cap_a*



Crides de BSD sockets: `recvfrom`

- Recull dades rebudes per un socket de tipus *no-connectat*(DGRAM). Retorna el nombre de octets rebuts (no es pot superar *#_octets*) o -1 en cas d'error. A *de* posa l'adreça de l'origen.

`recvfrom`

```
int recvfrom(int sockfd, char *buff, int #_octets, int flags, struct sockaddr *de, int *longitud);
```

- *sockfd* = descriptor de socket (de `socket()`)
- *buff* = dades a rebre
- *#_octets* = longitud màxima a rebre
- *flags* = paràmetres especials, generalment 0 (MSG_OOB, MSG_DONTROUTE, etc.)
- *de* = adreça origen
- *longitud* = longitud en octets de *de*



Crides de BSD sockets: `close`

- Tanca el socket especificat.

`close`

```
int close(int sockfd);
```

- *sockfd* = descriptor de socket (de `socket()`)
- Per controlar de millor forma el tancament disposem de la crida `shutdown()`



Crides de BSD sockets: Altres crides

shutdown

```
int shutdown(int sockfd, int flags);
```

getpeername

```
int getpeername(int sockfd, struct sockaddr *adr, int *longitud);
```

getsockname

```
int getsockname(int sockfd, struct sockaddr *adr, int *longitud);
```

gethostname

```
int gethostname(char *nomhost, size_t mida);
```



Crides de BSD sockets: Altres crides

select

```
int select(int numdf, fd_set *dflect, fd_set *dfescl,  
fd_set *dfexce, struct timeval *tempor);
```

Macros:

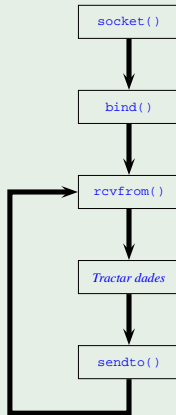
```
void FD_ZERO (fd_set *set);  
void FD_SET (int filedes, fd_set *set);  
void FD_CLR (int filedes, fd_set *set);  
int FD_ISSET (int filedes, const fd_set *set);  
int FD_SETSIZE;
```



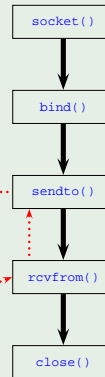
Estructures client/servidor

Interactiu UDP

Servidor interactiu



Client



Red dotted arrows indicate communication between the server and client:

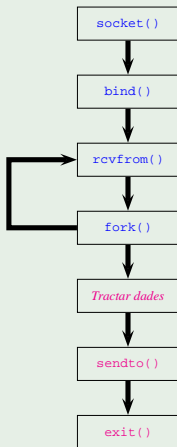
1. From Client `sendto()` to Server `rcvfrom()`.

2. From Server `sendto()` to Client `rcvfrom()`.

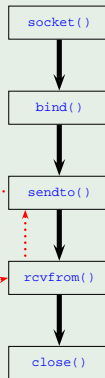
Estructures client/servidor

Concurrent UDP

Servidor concurrent



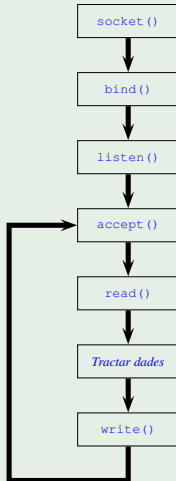
Client



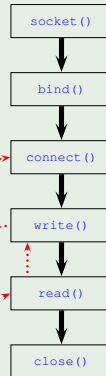
Estructures client/servidor

Interactiu TCP

Servidor interactiu



Client



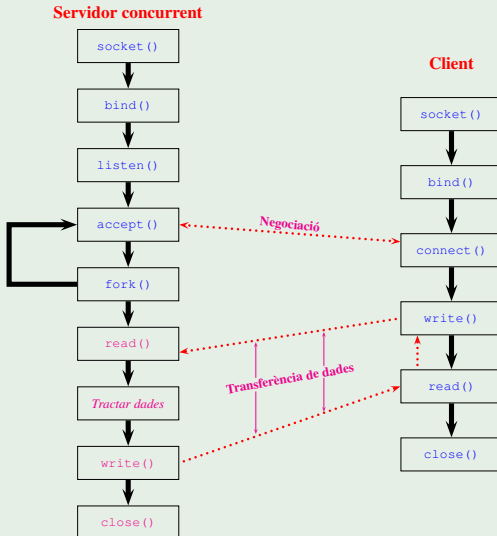
Negociació

Transferència de dades



Estructures client/servidor

Concurrent TCP



Python: Mòdul socket

Conté funcions bàsiques de treball amb sockets (així com la creació de sockets).

- `getfqdn(ipaddr)`: Retorna el Fully Qualified Domain Name per una màquina. Si és `' '` retorna el nostre nom.
- `gethostbyaddr(ipaddr)`: Retorna una tupla amb `(hostname, alias_list, ipaddr_list)`. On hi haurà el nom primari, la llista d'àlies i la llista d'IPs dels alias i nom.
- `gethostbyname_ex(hostname)`: El mateix que `gethostbyaddr`, però accepta un nom de màquina o una IP (en quad) com paràmetre.



Python: Mòdul socket

- `htonl(i32)`, `htons(i16)`: Converteix `int32` o `int16` de host a màquina.
- `ntohl(i32)`, `ntohs(i16)`: Converteix `int32` o `int16` de màquina a host.
- `inet_aton(string)`: Converteix adreça IP (en text) a packed string de 4 bytes
- `inet_ntoa(string)`: Converteix adreça IP (en packed string de 4 bytes) a cadena.



Creació de sockets: `socket.socket()`

```
socket(familia, tipus)
```

La crida `socket` crea i retorna un objecte de tipus `socket`. Tipus i família són similars als de les crides de BSD Sockets per C: `AF_INET`, `AF_INET6` o `AF_UNIX` per una banda i `SOCK_STREAM` i `SOCK_DGRAM` per un altra.

Disposen també de: `SOCK_RAW`, `SOCK_RDM` (UDP confiable) i `SOCK_SEQPACKET` (UDP confiable i orientat a connexió).

Creació de socket

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```



Objectes socket (I)

La classe que modela els sockets (`socket.SocketType`) i es retornada per la crida `socket.socket()` té totes les funcions bàsiques de treball en sockets similars (idèntiques de fet) a les de BSD Sockets per C.

Un objecte `s` de tipus `socket` proporciona:

- `s.bind(host, port)`: vincula un socket a un port per escoltar. `host` pot ser cadena buida: "", i vol dir qualsevol de les nostres adreces IP.
Només es pot cridar un cop per socket.
- `s.listen(backlog)`: Estableix el socket a escoltar connexions i dona la mida de la cua de connexions en espera.
- `s.connect((host, port))`: Connecta el socket cap al `host` `host` i el port `port`. Crida bloquejant, i en cas d'error llença excepcions.



Objectes socket (II)

- `(s1, (ip, port)) = s.accept()`: accepta una connexió pendent i retorna una parella: `s1` és el nou socket i `(ip, port)` la parella de l'adreça IP de l'altre extrem de la connexió i el port. `s` ha de ser `SOCK_STREAM` i prèviament s'ha d'haver cridat `bind` i `listen`. Si no hi ha clients intentant connectar, la crida es bloqueja.
- `s.close()`: Tanca el socket. No es pot cridar més mètodes a `s` després.
- `f = s.makefile([mode[, bufsize]])`: Crea un objecte `f` de tipus `file` associat al socket `s`. Podem aleshores emprar-lo com un fitxer normal. Els paràmetres són els mateixos que al crear fitxers.
Python tanca el socket només quan l'objecte `s` i el fitxer `f` s'han tancat.



Objectes socket (III)

- `str=s.recv(size)`: Rebrà com a màxim `size` bytes del socket i els retorna (cadena). Retorna cadena buida en cas de desconnexió. Si no hi ha dades el socket es bloqueja.
- `(data,(ip,port))=s.recvfrom(size)`: Rebrà com a màxim `size` bytes del socket i retorna una tupla on: `data` és la cadena de màxim `size` bytes i `(ip,port)` són l'adreça IP i el port de l'emissor. Crida bloquejant.

Emprat en sockets no orientats a connexió (UDP).



Objectes socket (IV)

- `n=s.send(string)`: Envia `string` pel socket `s` i retorna el nombre de bytes enviats. Pot no enviar-los tots, aleshores hem de reenviar `string[n:]`.
Si no hi ha espai als buffers la crida es bloquejant.
- `s.sendall(string)`: Envia `string` pel socket `s` de forma completa, si no pot enviar `string` íntegrament bloqueja fins aconseguir-ho.
- `n=s.sendto(string, (host,port))`: Envia `string` pel socket `s` a `(host,port)`. Retorna el nombre de bytes enviats, podent no enviar-se tots. Es tasca nostra enviar aleshores `string[n:]`.
Emprat en sockets de datagrama (UDP).



Mòdul select

Conté una implementació independent de plataforma de `select()` (i altres utilitats depenents del sistema operatiu).

select

```
i,o,e = select(inputs, outputs, excepts,  
timeout=None)
```

- `inputs`, `outputs` i `excepts` son llistes de sockets a ser revisades per si poden llegir-se (`inputs`), escriure's (`outputs`) o hi ha hagut excepcions (`excepts`).
- `timeout` és un nombre (float) de segons a esperar, si és 0 retorna immediatament sense esperar esdeveniments i si es `None` espera indefinidament.
- Retorna una tupla (`i,o,e`) on `i` és una llista de 0 o més elements sockets d'on podem fer una lectura no bloquejant. `o` és una llista de sockets on podem escriure. I `e` és una llista de sockets on hi ha hagut una excepció.



Mòdul `asyncore`

`Asyncore` (i `asynchat`) permeten implementar servidors i clients asíncrons de forma fàcil i més eficient que `select`.

```
asyncore.loop()
```

Implementa el bucle principal (implementat internament per `asyncore`).

Per implementar el que volem hem de crear classes derivades de `asyncore.dispatcher` o `asyncore.dispatcher_with_send`.

```
d.create_socket(family, type) crea el socket per  
asyncore.dispatcher
```

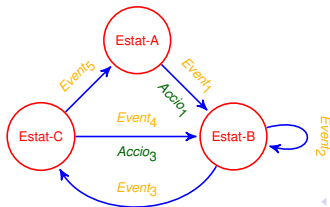
En les classes derivades hem d'implementar els `handle_` pertinents:
`handle_accept`, `handle_close`, `handle_connect`,
`handle_read`, `handle_write`.



Protocol (I)

Conjunt de regles i convencions que controlen l'intercanvi de missatges entre dues entitats d'una xarxa de comunicacions.

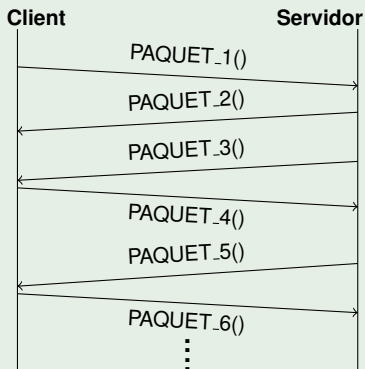
- Un protocol pot ser vist com una màquina d'estats. Molts cops es dissenyen i s'especifiquen (fins i tot es programen) com a tals.
- Per definir el comportament d'un protocol s'empra el **Diagrama d'estats**.
- Un diagrama d'estats té tres components:
 - Estats: Situació estable del protocol.
 - Events: Esdeveniment que pot comportar un canvi d'estat.
 - Accions: Resposta a un event.



Protocol (II)

- Per especificar el funcionament de l'intercanvi de missatges s'empren els diagrames de temps:

Diagrama de temps



Protocol (III)

- Cada protocol té una estructura o format de paquets pròpia (PDU, *Protocol Data Unit*).
- De forma general, un paquet consta de:
 - Un camp d'informació (payload): el que volem transmetre.
 - Camps de capçalera/cua (header/tail) . Ubicats davant i/o darrera de la informació

PDU

Adreça origen

Adreça destí

Tipus paquet

Informació o dades

Comprovació d'errors

