Programació de sockets en Java

PID_00275866

Maria Isabel March Hermo

Temps mínim de dedicació recomanat: 5 hores

Universitat Oberta de Catalunya

Maria Isabel March Hermo

L'encàrrec i la creació d'aquest recurs d'aprenentatge UOC han estat coordinats pel professor: Joan Manel Marquès Puig

Primera edició: setembre 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Autoria: Maria Isabel March Hermo
Producció: FUOC

Índex

Introducció

La raó de ser de l'existència d'Internet és la comunicació entre dispositius de qualsevol índole, connectats arreu del món. Actualment hi ha una infinitat d'aplicacions en xarxa, de temàtiques molt diferents: web, correu electrònic, navegació GPS, reproducció via *streaming* d'àudio i vídeo, jocs multiusuaris, comerç electrònic, electrodomèstics intel·ligents, videotutorials, xarxes socials...

Aquesta xarxa estructurada, complexa i hetereogènia, constantment vehicula els intercanvis de missatges que es produeixen entre els processos que executen aquestes aplicacions, de manera transparent, a l'usuari final.

En aquest mòdul aprendrem a programar una d'aquestes aplicacions en xarxa, mitjançant la interfície de programació d'aplicacions (API) més popular per a comunicar processos remots: els *sockets*.

Quan estem comunicant dues aplicacions de dispositius diferents o més necessitem una sèrie de convencions perquè la xarxa pugui redirigir els paquets a una i altra banda. Els *sockets* són la interfíce programari (*software*) per a crear aquest marc comunicatiu, és a dir, un conjunt d'instruccions que ha de seguir l'aplicació per comunicar-se amb el nivell de transport i els seus homòlegs en el destí, i així poder permetre l'intercanvi de dades entre processos locals o remots.

Hi ha diverses classificacions de *sockets*, però s'aprofundirà especialment en la tipologia segons el servei que ofereixen: orientats o no a la connexió o, el que és el mateix, basats en el protocol de transport TCP o UDP. Aquesta diferenciació és clau, ja que defineix la naturalesa dels *sockets*, com actuen i com es comporten al llarg de la comunicació. En conseqüència, la seqüència d'operacions que s'hauran de realitzar en un o altre cas és diferent.

Després de definir el marc conceptual, el mòdul s'endinsa dins les classes en Java que donen suport a la programació de *sockets*. Es veuen en detall els constructors, els mètodes principals i les excepcions que poden llançar-se, pel la qual cosa es pot usar com a manual de referència en la programació d'aplicacions en xarxa.

Si bé no calen coneixements avançats de la programació del llenguatge Java, sí que és recomanable saber com es duu a terme la compilació i execució d'arxius en aquesta plataforma, així com els fonaments de la programació estructurada (classes, constructors, mètodes *getters* i *setters*...) i la sintaxi dels tipus de dades primitius més rellevants i les instruccions bàsiques per a manipular-los.

Objectius

Els objectius principals que l'estudiant ha d'assolir en la comprensió d'aquest mòdul didàctic són els següents:

1. Definir la interfície de programació *sockets* i situar-la dins el marc conceptual de la pila de protocols TCP/IP, el paradigma client-servidor i els diferents mecanismes de comunicació que hi ha.

2. Diferenciar els trets característics de les comunicacions orientades i no orientades a la connexió i saber escollir la idònia segons la funció final de l'aplicació en xarxa.

3. Dibuixar el diagrama de flux típic d'un client i d'un servidor orientat i no orientat a la connexió.

4. Conèixer les classes Java relacionades amb la programació de *sockets*, orientats o no a la connexió. En particular, els seus constructors, els mètodes principals i els seus paràmetres.

5. Programar una aplicació en xarxa, comunicant processos remots via *sockets*, tant en el rol dels clients com en el rol dels servidors.

1. Història

El terme *connector* o *socket*, entès com a mecanisme de comunicació entre processos remots, va sorgir l'any 1971, en RFC 147 «The Definition of a Socket», escrit per J.M. Winett.

> «A socket is defined to be the unique identification to or from which information is transmitted in the network. The socket is specified as a 32 bit number with even sockets identifying receiving sockets and odd sockets identifying sending sockets. A socket is also identified by the host in which the sending or receiving processer is located.»

Aquest document no era públic, sinó utilitzat dins la xarxa ARPANET, considerada com la precursora de les xarxes que s'usen avui dia. Inicialment era una xarxa de caràcter militar creada pel Departament de Defensa dels Estats Units al final dels anys seixanta del segle passat, i que va acabar connectant moltes universitats i instal·lacions governamentals utilitzant línies telefòniques convencionals. Més endavant, quan es van afegir enllaços per satèl·lit o ràdio, els sistemes van començar a tenir problemes per a interactuar amb aquestes noves xarxes. Es va fer palès que calia una nova arquitectura de referència per poder connectar diferents models de xarxes. Aquesta arquitectura es va popularitzar com el model de referència TCP/IP (inicials dels seus dos principals protocols), que és el model d'Internet. Els *sockets* sorgeixen arran de la necessitat de connectar els nivells superiors d'aquest model TCP/IP i, per una altra banda, de comunicar aplicacions que s'estaven executant en dispositius heterogenis de la xarxa ARPANET.

No obstant això, avui en dia, la definició de *sockets* dista molt de la idea inicial definida al RFC 147, tot i que la seva funció és la mateixa. La majoria d'implementacions dels *sockets* es basen en els *Berkeley sockets*, anomenats així, perquè agafen com a referència una distribució del sistema operatiu Unix, en la variant que va fer la Universitat de Berkeley, versió 4.2. (UNIX BSD 4.2.), l'any 1983. En aquesta distribució, es van implementar un seguit de crides a sistema que implementaven la interconnectivitat entre processos via *sockets*, aglutinades en una interfície de programació d'aplicacions específica (*sockets API*). Aquestes aplicacions, empraven la pila de protocols TCP/IP i eren testades a la xarxa ARPANET i la seva successora ARPA.

Segons l'estàndar de Berkeley, els *sockets* eren implementats com un tipus de descriptor de fitxers, seguint la filosofia Unix. Per això, al treballar amb *sockets* trobarem funcions similars als clàssics `open()`, `read()`, `write()` o `close()`, més pròpies de l'àmbit de fitxers. Actualment, la interfíce de programació via *sockets* s'ha adaptat als diferents sistemes operatius. Per exemple, *WinSock* és una altra implementació basada en els Berkeley *sockets*, usada pel sistema operatiu Microsoft.

Hi ha altres implementacions similars a l'API de *sockets* que defineixen un conjunt de crides a sistema per a comunicar-se amb els protocols de transport. Una d'elles és la interfície TLI (*Transport Layer Interface*), basada en fluxos o *streams* de dades, distribuïda en una altra variant de UNIX anomenada System V, en la seva versió 3 (AT&T UNIX System V o SVR3), l'any 1987. La intefície TLI està basada en el model de referència OSI (*Open System Interconnection*), que va quedar obsolet davant el model TCP/IP.

2. Què és un *socket*?

En l'apartat anterior hem vist què va motivar la necessitat de definir una interfície de programació via *sockets*, però, què són exactament els *sockets*?

L'analogia clàssica que es fa servir per a entendre el concepte de *sockets* és el sistema telefònic com a mitjà per a comunicar persones. Els *sockets* serien els telèfons, que permeten intercanviar informació entre processos, que serien les persones. Així doncs, els *sockets* són punts de comunicació entre agents (processos o persones respectivament), mitjançant els quals es pot enviar o rebre informació.

Els *sockets* són mecanismes de comunicació entre processos, locals o remots, que permeten que un procés es comuniqui bidireccionalment amb un altre procés.

Una altra analogia la trobem en el servei postal. Quan volem enviar una carta a una altra persona, primer hem d'escriure'n el contingut (les dades) i posar-la dins d'un sobre. Després de tancar-lo, haurem d'escriure el nom complet del destinatari, l'adreça i el codi postal, normalment a la part dreta inferior del sobre. Després posarem un segell a la part dreta superior; i finalment portarem la carta a una bústia o una oficina de missatgeria. De la mateixa manera que el servei postal té aquest conjunt de regles que ha de seguir l'emissor de la comunicació per a assegurar-se que el destí rep la carta, Internet fa ús de les interfícies *sockets* perquè el procés que envia les dades ho faci segons les normes establertes.

Els *sockets* són una interfície de programari formada per un conjunt d'instruccions de codi, que els programes que es comuniquen per la xarxa han de seguir, perquè Internet pugui entregar les dades.

2.1. Els *sockets* en el model de referència TCP/IP

Els estàndards de comunicacions en xarxa proporcionen la base per a la transmissió de dades entre diferents equips, per a la fabricació d'equips de xarxa compatibles i per al disseny de les rutines dins els sistemes operatius que facilitin les comunicacions a distància.

Hi ha dos models de comunicació principals que utilitzen una estructuració basada en nivells: el model de referència OSI (*Open System Interconnection*), que ha quedat relegat a l'àmbit teòric, i el model de protocols TCP/IP (*Transport Control Protocol/Internet Protocol*), àmpliament utilitzat i, per això, anomenat també model d'Internet o pila de protocols d'Internet.

Recordem que la pila de protocols Internet s'estructura en cinc nivells, tal com es mostra a la taula següent.

Taula 1.

Aplicació	***Dades***	***DNS, HTTP, P2P, EMAIL, TELNET, FTP***
Transport	***Segments/Datagrames***	***TCP, UDP***
Xarxa	***Paquets***	***IP, ARP, ICMP***
Enllaç	***Trames***	***ETHERNET, 802.11, ATM, PPP***
Físic	***Bits***	***RS-232, RJ-45, DSL, 100BASE-TX***

Cada nivell té una funció diferenciada, que resumirem a continuació. A més, en cadascun podem trobar diferents protocols que responen a un d'aquests nivells serveis particulars i diferenciats del nivell on es troben.

- **Físic**: processa la transmissió de bits pel mitjà físic de transport, com fibra òptica, coaxial, radio...

- **Enllaç**: permet la transmissió de trames entre màquines connectades directament. La xarxa més usual és Ethernet.

- **Xarxa**: defineix l'adreçament i l'encaminament (*routing*) independent de paquets per la xarxa, de manera que arribin de l'origen al destí seguint la millor ruta. El protocol més utilitzat és IP. També hi trobem ARP (resolució d'adreces) o ICMP (missatges de control).

- **Transport**: s'encarrega de proporcionar serveis a la comunicació extrem a extrem, com la fiabilitat i la seguretat que les dades arribin al destí en l'ordre correcte. El protocol orientat a la connexió TCP o el no orientat a la connexió UDP s'utilitzen segons les necessitats de les aplicacions.

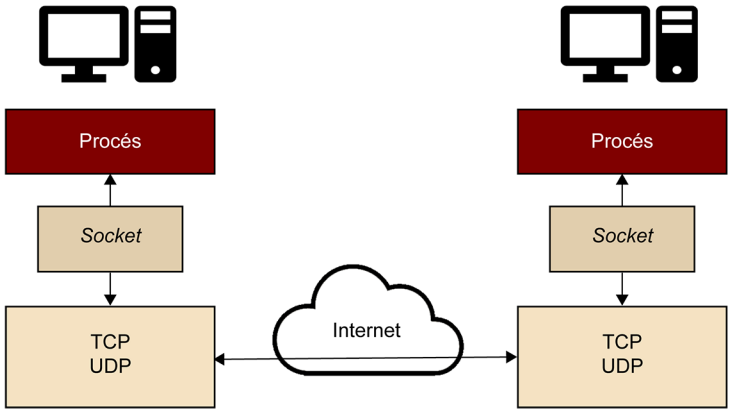
- **Aplicació**: és la més propera a l'usuari. Comprèn les aplicacions i els processos que utilitzen les xarxes de comunicacions. Els protocols més populars són HTTP (*Hypertext Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*) o SSH (*Secure Shell*), entre d'altres.

La comunicació entre nivells té dues vessants:

- Extrem a extrem de la comunicació respecte del mateix nivell.

- Nivell immediatament inferior i superior respecte del mateix dispositiu.

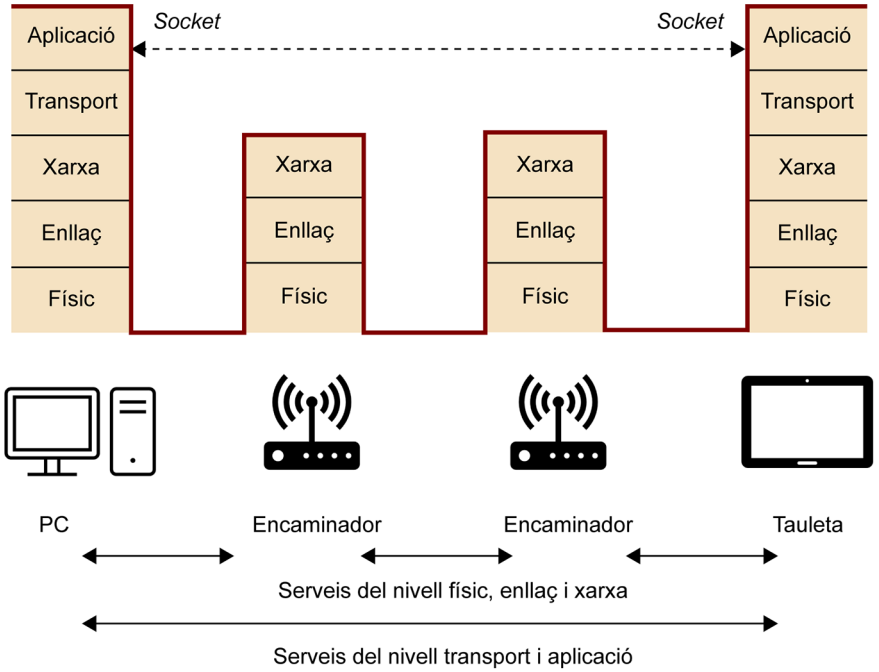
Figura 1.



Com pot observar-se, els *sockets* es troben entre el nivell d'aplicació i el nivell de transport. També s'anomenen API (*Application Programming Interface*) entre l'aplicació i la resta de nivells que ens comunicaran per la xarxa.

El nivell d'aplicació, concretament a partir dels processos que són realment els artífexs actius de les comunicacions, envia mitjançant *sockets* les dades a la capa de transport, que després de l'anàlisi i la gestió pertinents, les traslladarà als nivells inferiors. Les dades s'enviaran per la xarxa al *host* destí, on finalment, després de l'anàlisi i la gestió pels nivells inferiors, arribaran al nivell de transport, des d'on es redirigiran al procés corresponent mitjançant el *socket*. Després de tractar les dades, és habitual que es doni el procés invers. Aquest procés s'anomena **encapsulament i desencapsulament de paquets** entre els nivells del model d'Internet. Podem observar aquest procés com la línia marró fort de la següent figura.

Figura 2.



Els *sockets* són interfícies de programació d'aplicacions (API) en xarxa que se situen entre el nivell de transport i el nivell d'aplicació en el model de referència TCP/IP.

Els elements intermedis de la xarxa, com els encaminadors (*routers*, en anglès) i els commutadors (*switches*, en anglès) no necessiten implementacions dels *sockets*, ja que ells operen en el nivell d'enllaç (*switches*) o en el nivell de xarxa (*routers*). No obstant, els tallafocs (*firewalls*, en anglès) els traductors d'adreces IP o els *proxies*, sí que fan un seguiment dels *sockets* actius. També en certes polítiques de qualitat de servei (QoS) implementades en els *routers* o certs protocols d'encaminament, s'empra informació sobre els *sockets*.

2.2. Els *sockets* en el paradigma client-servidor

Fins al moment, hem parlat de com les aplicacions es comuniquen entre sí per la xarxa. Hi ha diverses maneres de possibilitar aquesta comunicació extrem a extrem a través d'Internet. La gran majoria d'aplicacions en xarxa, que empren els *sockets* per a comunicar-se, segueixen el paradigma client-servidor.

L'arquitectura client-servidor és un model de disseny de programari basat en dos perfils que es comuniquen:

- Un proveïdor de recursos o serveis, els anomenats **servidors**.
- Els demandants d'aquestes serveis, els anomenats **clients**.

Els programes clients s'executen en *hosts* que normalment interactuen amb els usuaris. Els processos creats com a conseqüència de l'execució d'aquests programes realitzen peticions a un servidor o diversos per a obtenir uns determinats serveis. Aquesta idea es pot aplicar de manera local, però és més comú i útil en sistemes distribuïts, a través d'una xarxa de computadors, on molts usuaris es comuniquen amb un servidor.

Els programes servidors són programes que s'estan executant en un *host* remot i ofereixen serveis a múltiples clients potencials, normalment de manera ininterrompuda i continuada.

Per exemple, en les aplicacions Web, un navegador client intercanvia missatges amb un domini allotjat en un servidor Web. En un sistema P2P de compartició d'arxius com BitTorrent, un fitxer es transfereix entre dos *hosts*: el que ofereix l'arxiu actúa com a servidor i el que es descarrega l'arxiu actúa com a client.

També es poden donar casos en què un determinat *host* actuï com a client i servidor alhora, per diverses aplicacions o fins i tot dins la mateixa aplicació. Per exemple, en BitTorrent també és comú que des d'un extrem s'estigui oferint un fitxer i, alhora, es descarreguin arxius oferts per altres.

En qualsevol dels dos rols, els *sockets* possibiliten la comunicació entre les dues parts.

- A la banda dels processos **servidors**, els *sockets* serveixen per a anunciar a Internet els serveis que s'ofereixen i possibilitar la comunicació amb els clients que els demanin.

- A la banda dels processos **clients**, els *sockets* serveixen per a comunicar-se amb els servidors, fent peticions de serveis i obtenint-ne les seves respostes.

2.3. Els *sockets* com a mecanisme de comunicació

Hi ha diverses tècniques i paradigmes per a realitzar la comunicació entre processos que s'executen de manera distribuïda, paral·lela o concurrent. Els màxims exponents en què es basa la gran part del programari en xarxa avui en dia són els següents:

- El **pas de missatges**: sincronitzar dos processos de manera que es passin cooperativament un missatge, de tal manera que un l'envia i l'altre el rep.

- La **invocació remota**: executar un procediment o mètode, normalment en un altre dispositiu, com si s'estigués fent a la mateixa màquina.

De fet, els dos són similars, ja que hi ha un procés actiu que envia el missatge o crida l'execució d'un mètode, respectivament. No obstant això, la invocació remota no exigeix que el receptor estigui a l'espera de llegir cap petició i, a més, la resposta és atòmica.

Els *sockets* se situarien com una implementació del **pas de missatges**, ja que hi ha diversos participants que es comuniquen bidireccionalment de manera explícita mitjançant el pas de dades.

2.4. Identificació dels *sockets*

Recuperarem un moment l'exemple anterior de la tramesa postal. Un dels punts imprescindibles si volem enviar una carta és saber l'adreça completa del destinatari. Sense aquestes dades, a l'empresa de transport li seria impossible fer l'entrega correctament. També se sol posar el remitent de la carta, per saber qui l'ha enviada i així poder respondre o comunicar un possible error en l'entrega. Aquestes dades es posen en l'anvers i revers d'una carta en un ordre concret:

- el nom complet,
- l'adreça amb el carrer i el número,
- el codi postal i la població.

De la mateixa manera, els processos que es volen comunicar per la xarxa necessiten identificar-se amb unes dades bàsiques. En l'argot de les xarxes, aquest identificador dels processos que es comuniquen per Internet és el **nom del *socket*** i està format per:

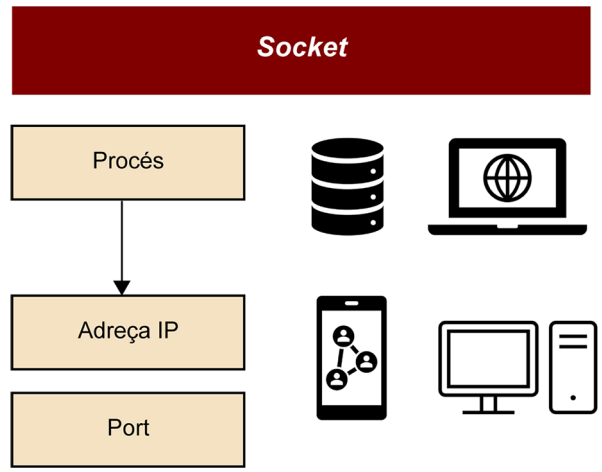
- **Adreça IP:** identifica qualsevol terminal de qualsevol xarxa del món a Internet. Són 4 bytes que se solen escriure en notació decimal: 4 nombres de 0..255 separats per punts.

- **Port:** identifica un procés concret en una màquina. Són 2 bytes que se solen escriure en notació decimal (un nombre). Per tant un port pot estar numerat del 0 al 655436.

No hem de confondre l'adreça IP amb l'adreça física, que és la adreça que identifica unívocament la targeta de xarxa de la màquina. Per exemple, en les xarxes Ethernet és l'adreça MAC, que es composa de 6 bytes que se solen expressar en hexadecimal, com per exemple 54:D4:6F:AD:67:E4. A diferència de l'adreça IP i el port, el programador no coneix les adreces MAC ni hi treballa, a no ser que sigui a baix nivell per a configurar algun aspecte específic relatiu al nivell d'enllaç.

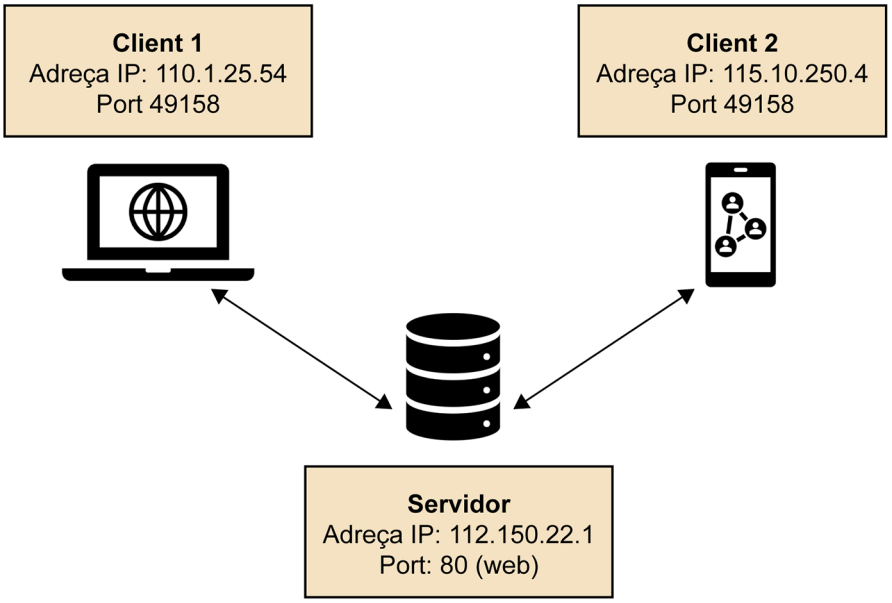
La identificació de cada *socket* per la seva adreça IP i port és única. L'adreça IP identifica el dispositiu dins la xarxa Internet i el port identifica el procés dins aquest dispositiu, sigui del tipus que sigui.

Figura 3.



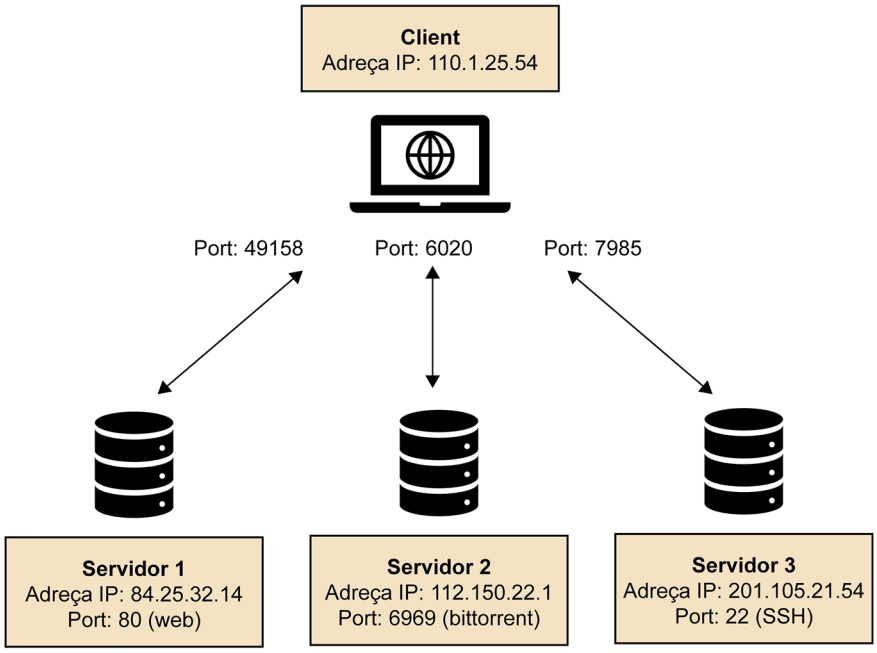
Per simplificar, ens posarem en el cas d'un ordinador amb una única targeta de xarxa. Aquest ordinador està identificat a la xarxa per la IP 80.55.23.69. Tots els processos que s'estiguin executant al mateix ordinador tindran aquesta mateixa IP. No obstant això, no poden haver-hi dos *sockets* oberts en un mateix moment a la mateixa màquina, amb el mateix número de port. Si no, el nivell de transport no seria capaç d'entregar els segments que li arriben a un procés o altre, ja que no tindria manera de diferenciar-ho. En canvi, sí poden haver-hi dos processos que s'estiguin comunicant per la xarxa amb el mateix número de port, sempre que pertanyin a equips diferents. És a dir, poden haver-hi dos processos amb el mateix número de port però amb adreces IP diferents. Així, el *socket* està identificat per aquest <adreça IP, port> de manera unívoca igualment, com es pot veure a la figura següent.

Figura 4.



Per exemple, en un mateix ordinador podem tenir obert un navegador amb diverses pestanyes obertes consultant webs, també ens estem descarregant uns arxius amb el programa BitTorrent o pujant d'altres de manera segura per SSH. Cadascuna d'aquestes aplicacions haurà creat com a mínim un *socket* per on s'intercanvien les dades amb el servidor pertinent, com es pot veure a la figura següent:

Figura 5.



Una determinada màquina podrà tenir un conjunt de servidors actius i connexions amb servidors remots mitjançant la distinció de ports.

A continuació, veurem amb més detall quines adreces IP i ports s'utilitzen per a programar les aplicacions en xarxa.

Adreces IP

En l'RFC 1918 al 1996 es defineixen els conjunts d'adreces IP destinades a ús intern i a ús públic:

- Les **adreces privades** són les les adreces IPv4 que es destinen a la creació de xarxes privades, com pot ser la xarxa interna d'una empresa que no necessita que es pugui accedir als equips directament des d'Internet.

- Les **adreces públiques** es veuen a tot Internet i, en conseqüència, no hi poden haver dues màquines amb la mateixa adreça IP.

Taula 2.

Classe	***Prefix***	***Rang***
A	10.0.0.0/8	10.0.0.0 – 10.255.255.255
B	172.16.0.0/12	172.16.0.0 – 172.31.255.255
C	192.168.0.0/16	192.168.0.0 – 192.168.255.255
La resta d'adreces són públiques		

L'assignació d'adreces IP als *hosts* se sol fer pel nostre ISP o proveïdor d' Internet. Com que el nombre d'adreces IP és limitat, els ISP van rotant aquestes adreces, per això mateix s'anomenen **adreces IP dinàmiques**.

A més, per no haver de recordar o anotar les adreces IP concretes dels servidors a què ens volem connectar, es van idear els **noms de domini** que utilitzem constantment avui en dia, com *uoc.edu* o *google.com*. Mitjançant el protocol DNS (*Domain Name Server*) es fa la traducció de nom de domini a adreça IP e inversa, de manera transparent a l'usuari. El programador, també té al seu abast funcions de conversió per a facilitar aquesta traducció a l'hora de crear un *socket* o connectar-se a un servidor.

Ports

Per una altra part, l'assignació de ports a processos, tampoc és arbitrària. L'associació IANA (*Internet Assigned Number Authority*) va definir tres rangs de ports:

- **Ports coneguts** (0..1023): ports fixos universals i coneguts, assignats per la IANA, associats a serveis específics que estan estandarditzats i regulats per un RFC.

- **Ports registrats** (1024..49151): ports usats per aplicacions d'usuari.

- **Ports dinàmics o efímers** (49151..65535): ports usats de manera temporal perquè un procés client es pugui connectar amb un procés servidor.

Normalment, si volem accedir a un servidor conegut, haurem de crear un *socket* client que faci peticions a un *socket* servidor, per la qual cosa haurem d'indicar el nom de domini d'aquest servidor i el port conegut segons la taula que es mostra a continuació.

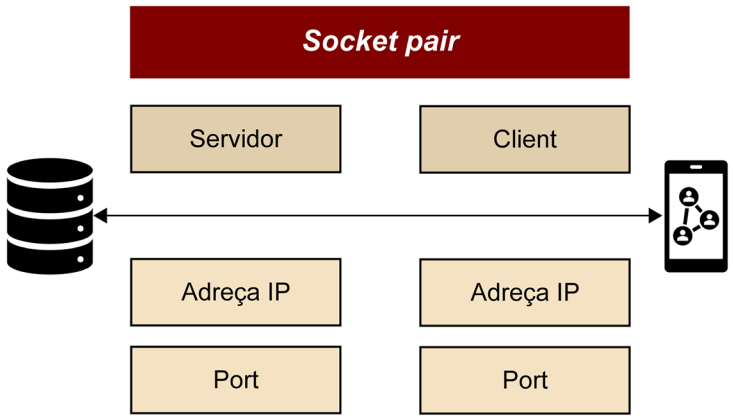
Taula 3.

Servei	***Port***	***TCP***	***UDP***
DayTime	13	√	√
FTP-Data	20	√	
FTP	21	√	
SSH	22	√	
Telnet	23	√	
SMTP	25	√	
DNS	53	√	√
HTTP	80	√	
POP3	110	√	
IMAP	143	√	
SNMP	161		√
HTTPS	443	√	
SIP	5060		√

2.5. Tipus de *sockets*

L'enllaç entre els dos *sockets*, el *socket* del procés servidor i el *socket* del procés client, permet la comunicació bidireccional, és a dir, que tots dos extrems de la comunicació poden escriure i llegir alhora. És el que anomenem ***socket pair***, i identifica una connexió unívocament. Com el seu nom indica està format pels *sockets* corresponents als dos extrems de la comunicació.

Figura 6.



Aquesta és una característica pròpia dels *sockets*, que els diferència d'altres mecanismes de comunicació com les *pipes*.

Taula 4.

Sockets	***Pipes***
Canals bidireccionals	Canals unidireccionals
Comunicació entre processos remots	Comunicació entre processos locals
Filosofia client-servidor	Simple intercanvi de dades

Un *socket pair* és un parell de *sockets*, normalment format pel *socket* client i el *socket* servidor, que es comuniquen entre si de manera bidireccional.

Si ens fixem en el rol del *socket*, podem distingir entre:

- ***Sockets* actius**: els que inicien la connexió amb l'altre extrem.
- ***Sockets* passius**: els que estan a l'espera de rebre connexions.

Per exemple, en una comunicació client-servidor típica, el servidor estarà a l'espera de connexions de clients potencials mitjançant un *socket* passiu. En canvi, el client s'intentarà connectar al servidor mitjançant un *socket* actiu.

Si ens fixem en el tipus d'aplicacions que utilitzen *sockets*, podem tenir:

- **Aplicacions estàndar**. Les que pretenen comunicar-se mitjançant un protocol estàndard segons les normes establertes en un RFC. Per exemple, volem programar un client que es comuniqui amb un servidor Web qualsevol. La comunicació via *sockets* d'aquest client haurà de seguir les normes establertes en l'RFC 2616 del protocol HTTP (Web), i haurà d'escriure pe-

ticions i d'interpretar respostes segons dicta el protocol. En cas contrari, el servidor Web no entendrà què li diu el client i li retornarà un error.

- **Aplicacions propiètaries**. Una altra opció és desenvolupar un client i un servidor propis, segons unes normes que ideem expressament per a aquesta comunicació, sense seguir cap estàndar. Aquesta opció és habitual si volem oferir un servei específic, que no està regulat encara, com pot ser un joc online, per exemple.

No obstant això, la classificació més important rau en el serveis de la comunicació oferts pel *socket*, ja que condiciona el tipus d'intercanvi que hi pot haver:

- *Sockets* orientats a la connexió.
- *Sockets* no orientats a la connexió.

Atesa la seva rellevància en la programació de comunicacions via *sockets*, la veurem en els següents apartats més a fons.

3. Serveis de la comunicació

Seguint amb l'analogia del servei postal, ens podem fixar ara en que totes les empreses repartidores solen oferir més d'un servei als seus clients per a l'entrega de cartes o paquets: entrega exprés, confirmació de recepció, seguiment de l'enviament realitzat, etc. De manera similar, Internet proporciona múltiples serveis a les seves aplicacions en xarxa.

En particular, quan estem desenvolupant un programa que es comunicarà per la xarxa, hem d'escollir quin protocol de transport ens interessa més utilitzar segons els serveis que necessitarem a l'hora de fer la supervisió de la trasmissió de les dades extrem a extrem: TCP o UDP.

Aquesta elecció ens condicionarà la programació en un tipus o altre de *sockets*, ja que defineix la naturalesa. A més, el tipus de comunicació que pot generar-se entre clients i servidors ha de ser la mateixa, és a dir, el parell de *sockets* client-servidor ha de ser TCP o UDP, ja que el protocol de transport és únic per a cada comunicació.

Repassem breument els serveis diferenciats que ofereixen aquests dos protocols del nivell de transport:

UDP (*User Datagram Protocol*) és un protocol no orientat a la connexió:

- No hi ha cap concepte de connexió establert entre els dos extrems de la comunicació.
- Cada datagrama que s'envia és independent de l'anterior i el següent.
- No hi ha fragmentació dels datagrames.
- No hi ha cap garantia sobre l'ordre de recepció dels datagrames ni tan sols si s'han rebut correctament. Aquests errors els ha de detectar l'aplicació i ha de retransmetre els datagrames si es considera necessari.

Aquestes característiques el converteixen en un protocol molt ràpid, molt utilitzat en consultes de petició i resposta puntuals a un servidor, i en aplicacions en què la rapidesa de la comunicació prima sobre l'exactitud de les dades. És molt utilitzat en aplicacions en temps real. Un exemple el tenim en la retransmissió d'una carrera de bicicletes online. L'usuari prefereix seguir la carrera ni que sigui amb uns breus talls o sense que la imatge tingui gran qualitat. També s'empra en protocols ràpids de petició-resposta, com DNS, emprat per a resoldre l'adreça IP d'un nom de domini.

TCP (*Transport Datagram Protocol*) és un protocol orientat a la connexió que ens proporciona:

- Fiabilitat en la transmissió dels segments.
- Fragmentació dels segments.
- Manteniment de l'ordre dels segments.
- Utilitza una suma de verificació (*checksum*, en anglès) per a detectar errors.
- Hi ha una negociació d'establiment de la connexió inicial (*handshake*).

Aquestes característiques el fan un protocol òptim per a aplicacions com el Web (HTTP) o SSH, en què necessitem que les dades transmeses per la xarxa arribin íntegrament al destinatari.

Segons estiguem implementant un servei que requereixi un protocol de transport o altre, la sèrie d'instruccions de codi que haurem d'utilitzar per a programar els *sockets* serà diferent.

Mantenint aquesta dualitat segons el protocol de transport, es defineixen principalment dos tipus de *sockets*:

- ***Sockets* en UDP o *datagram sockets*** *(*datagrames): per a comunicacions en mode no connectat, amb l'enviament de datagrames de mida limitada (tipus telegrama).

- ***Sockets* en TCP o *stream sockets*** (flux de dades): per a comunicacions fiables en mode connectat, de dues vies i amb mida variable dels missatges de dades.

En *sockets* TCP o *stream sockets*, la interfície de *sockets* distingeix clarament entre el rol client i el rol servidor, que estableixen una connexió permanent, bidireccional, unívoca i fiable mentre duri la comunicació. En canvi, en *sockets* UDP o datagram *sockets* no hi ha aquest concepte de connexió entre els dos extrems de la comunicació ni hi ha cap tipus de control d'errors dels datagrames. Si es vol oferir un servei amb un cert grau de fiabilitat, s'ha de programar explícitament.

També hi ha altres tipus de *sockets* més específics, com els ***raw sockets***, que permeten l'accés a protocols de més baix nivell com el IP (nivell de xarxa). Els *raw sockets* faciliten l'accés expert a les comunicacions i s'empren per a configurar certs paràmetres dels nivells inferiors de la pila de protocols TCP/IP, variant-ne el seu funcionament estàndar.

3.1. *Sockets* UDP o no orientats a la connexió

A continuació, veurem el funcionament dels *sockets* que utilitzen UDP com a protocol de transport. Recordem que, en ser un protocol no orientat a la connexió, no ofereix fiabilitat i, per tant, no hi ha un control del flux de dades ni tampoc fragmentació. La capa d'aplicació serà l'encarregada de donar les instruccions pertinents, si es vol cert grau de fiabilitat.

Aquesta rapidesa en les comunicacions, sense que hi hagi un establiment previ i continu de la comunicació, implica que cada un dels datagrames hagi d'incorporar informació d'adreçament a la seva capçalera.

En comunicacions no orientades a la connexió, cada datagrama ha d'incloure el *socket pair*, això és, l'adreça IP i port a qui va dirigit el datagrama, i l'adreça IP i el port del remitent.

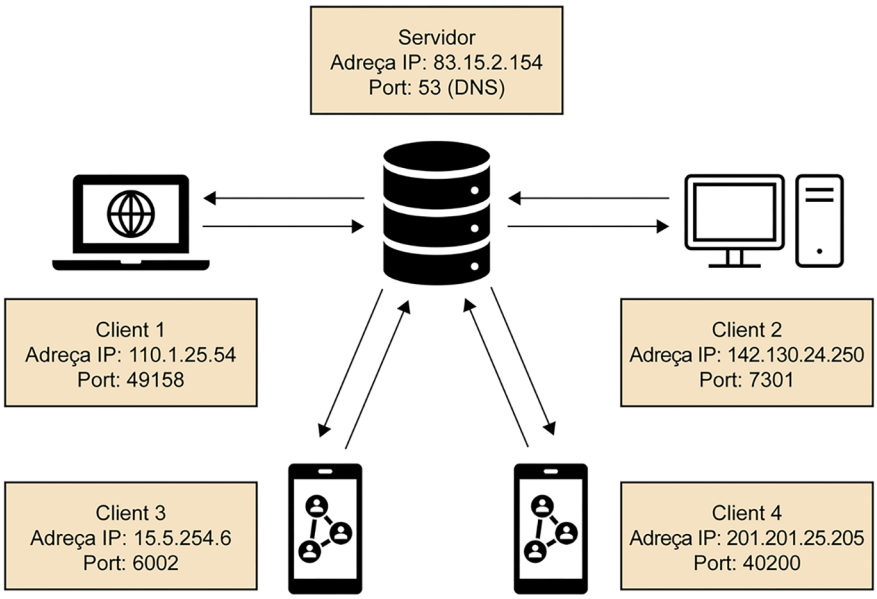
En cas que no hi hagués aquesta informació, els extrems de la comunicació no sabrien amb qui s'estan comunicant, perquè no hi ha una connexió establerta i mantinguda per on viatgen les dades. Cada datagrama que es rep es tracta individualment.

De fet, si recuperem l'analogia amb el servei postal, no és altra cosa que les dades que posem sobre el destinatari i el remitent d'una carta. Si una carta no té escrit el destinatari, el servei postal no la pot entregar. I si no té el remitent, qui rep la carta no tindrà l'adreça on dirigir la seva resposta i, per tant, no la podrà respondre.

En el paradigma client-servidor, els clients i el servidor es guien per un mecanisme petició-resposta.

En no haver-hi una connexió dedicada entre els dos extrems de la comunicació, el servidor haurà d'anar alternant l'atenció als múltiples clients que li facin peticions. És a dir, el servidor ha de tractar les peticions que li arribin, independentment del client que les faci, segons l'ordre d'arribada. Les ha d'analitzar i tractar, i ha de respondre al client destinatari que s'ha indicat en la mateixa petició.

Figura 7.



En les comunicacions no orientades a la connexió, hi ha un únic *socket* de nom conegut en el servidor, que està a l'espera de peticions de nous clients, i quan n'hi ha, les respon escrivint pel mateix *socket*.

Els processos clients també han de crear un *socket* sense nom conegut, que fan servir per a fer peticions amb el servidor directament, sense un establiment de la connexió prèviament.

El *socket pair* entre client i servidor no és, per tant, dedicat. Va variant en el transcurs de la comunicació i es manté únicament mentre dura aquest tractament de la petició i l'elaboració de la resposta cap a un determinat client.

Veiem aquest procés amb més detall:

- El servidor crea un *socket* d'escolta, amb nom conegut (adreça IP i port), per mitjà del qual el servidor espera les peticions de clients potencials.

- Cada client crea un *socket* sense nom per llançar les peticions al servidor.

- El client escriu les dades pel *socket* creat i indicat la petició realitzada i el *socket pair*, això és, l'adreça IP i el port del servidor, d'una banda, i l'adreça IP i el port seus (del client), de l'altra. En posar les dades de l'emissor de la comunicació, el servidor sap on ha de contestar.

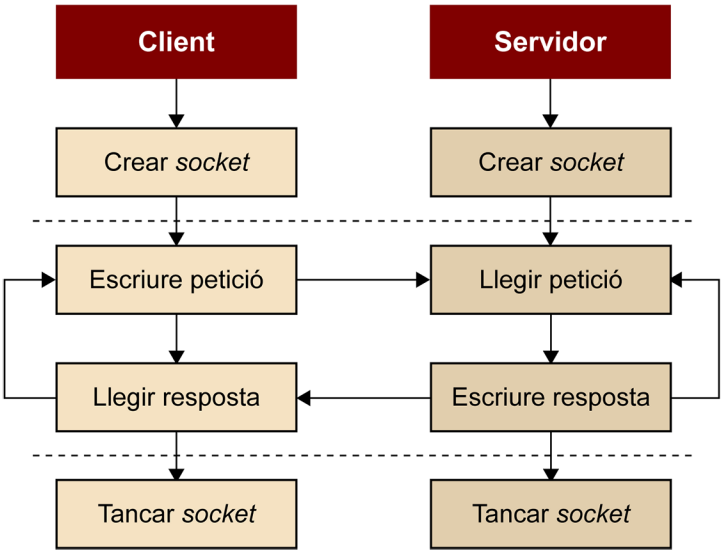
- El servidor rep la petició del client, llegint del *socket* d'escolta. L'analitza i la tracta, i escriu les dades relatives a la resposta en el mateix *socket*. Aquesta resposta també porta les adreces i ports del *socket pair*.

- Quan la comunicació acaba, el client tanca el seu *socket* per tal d'evitar el consum innecessari de recursos.

- El servidor continua atenent les peticions que li arribin de clients.

- El servidor tancarà el seu *socket* quan es pari, és a dir, quan ja no vulgui actuar més com a tal.

Figura 8.



En comunicacions no orientades a la connexió també es pot emprar concurrència (fils o *threads*) en el servidor. No obstant això, atés que és un protocol orientat a tractament de peticions i enviament de respostes ràpides, no és tan imprescindible.

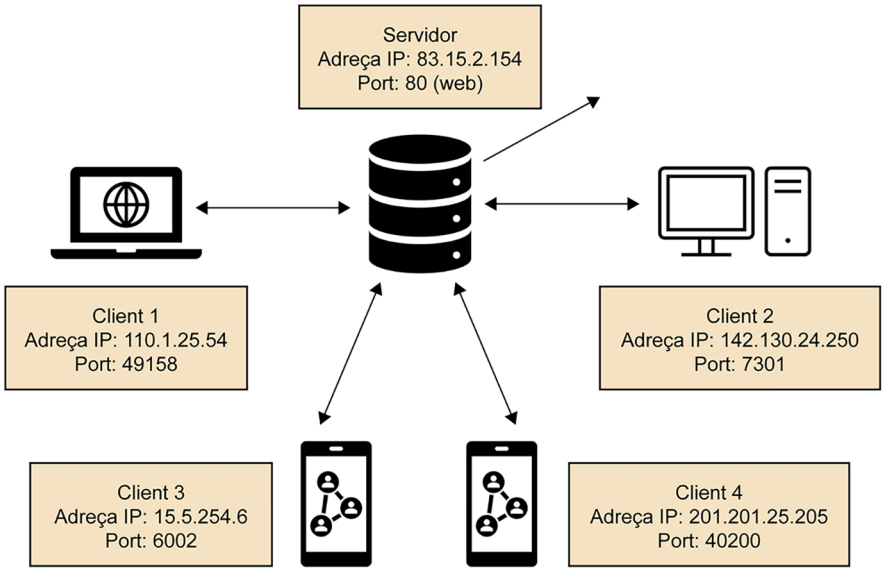
3.2. *Sockets* TCP o orientats a la connexió

Ara concretarem què ocorre durant la comunicació entre processos mitjançant *sockets* orientats a la connexió. Recordem que aquest tipus de *sockets* usen TCP com a protocol de transport i, per tant, és aquest protocol el que s'encarrega de lliurar els segments de manera fiable. Aquesta fiabilitat es tradueix en el fet que el flux de bytes que es genera en el procés origen es lliura sense errors al procés destí. A més, el protocol fragmenta el flux de dades procedent de la capa d'aplicació en missatges més petits.

Seguint el paradigma client-servidor, un procés actuarà de procés servidor creant un *socket* de nom conegut i oferint un servei. Aquest *socket* s'anomena *socket* d'escolta. Així els potencials clients podran connectar-s'hi per obtenir els serveis oferts.

D'altra banda, els processos clients també han de crear un *socket* sense nom conegut, que hauran de fer servir per comunicar-se amb el servidor al que es connecten. Aquest *socket* establirà una connexió amb el servidor. Com a conseqüència, es crea un nou *socket* sense nom en el servidor que ha de connectar únicament amb el client connectat.

Figura 9.



En comunicacions orientades a la connexió, tenim el següent:

- Un *socket* d'escolta en el servidor a l'espera de clients potencials.
- Un *socket* en el servidor per cada client que s'hagi connectat.
- Un *socket* en el client que el connecta amb el servidor en qüestió.

Aquests dos últims *sockets* són els que defineixen el *socket pair*, que vehicula l'intercanvi de missatges entre clients i servidor.

Hi pot haver tants *sockets pairs* en el servidors com clients estiguin connectats. En canvi, normalment només hi ha un *socket* d'escolta que espera connexions de potencials clients.

Ara que en tenim una idea general, vegem quin és el funcionament particular:

- El servidor crea un *socket* d'escolta, amb nom conegut (adreça IP i port), per mitjà del qual el servidor espera les connexions de potencials clients.

- Cada client crea un *socket* sense nom per llançar les peticions de connexió al servidor.

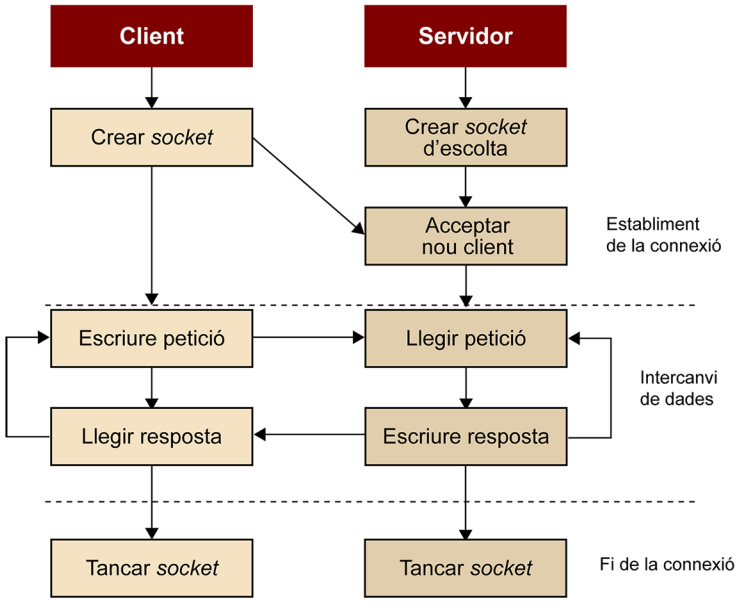
- El procés servidor accepta aquesta connexió creant un nou *socket* que serà el que realment mantindrà una comunicació bidireccional amb el client, i mitjançant el qual intercanviaran peticions i respostes. El *socket* amb nom creat inicialment se sol mantenir a l'espera de noves connexions d'altres clients.

- Un cop establerta la connexió, es tracta com un flux (*stream*) típic d'entrada i sortida, escrivint dades i llegint-les, tant en el client com en el servidor.

- Quan han acabat de comunicar-se, tant el *socket* client com el *socket* servidor, els dos sense nom conegut, s'han de tancar per donar per acabada la comunicació i evitar així el consum de recursos de sistema innecessaris.

- El servidor segueix a l'espera de peticions de connexió de nous clients mitjançant el *socket* d'escolta.

Figura 10.



Aquest és el procés típic que segueix en un servidor seqüencial. No obstant això, en les comunicacions en xarxa, és molt comú emprar **concurrència (fils o *threads*)** en el servidor, és a dir, que un procés *pare* creï altres processos *fills* per a atendre en paral·lel els clients que es van connectant. Així, un client no ha d'esperar que el servidor acabi d'atendre el client que s'ha connectat anteriorment, per tal de comunicar-s'hi, el temps d'espera conseqüent. El resultat de la qual cosa són múltiples comunicacions bidireccionals i simultànies que

es produeixen en paral·lel entre els fills del servidor i els clients. Aquest fet implica que hi hagi múltiples *sockets* creats en un mateix servidor, un d'escolta i un per cada client que es connecta.

4. Programació en Java

4.1. Conceptes bàsics

Ara que ja hem finalitzat l'exposició del marc conceptual, el portarem a la pràctica i crearem la nostra primera aplicació en xarxa.

Java és un llenguatge de programació creat en 1995 per l'empresa *Sun Microsystems*. El gran tret diferencial d'aquesta plataforma gratuïta és que és multiplataforma, és a dir, es va dissenyar amb la idea que els programadors escrivissin el programa amb independència del sistema operatiu o dispositiu en el què s'executés. Aquest fet és possible gràcies a la màquina virtual de Java (JVM), que s'encarrega d'aquesta portabilitat, fent-la transparent a programadors i usuaris finals. La rapidesa, robustesa, seguretat i fiabilitat fan que sigui ideal per a escriure fàcilment aplicacions en xarxa. A més, compta amb una comunitat a la xarxa molt gran que ens serveix per a consultar temes bàsics o específics de qualsevol aspecte de programació. Com a suport, els desenvolupadors de Java poden consultar la documentació online oficial de les seves API, també anomenada *Javadoc*.

Com qualsevol altre llenguatge de programació, Java té les seves pròpies regles i sintaxis, derivades del C i de la programació orientada a objectes (OOP). Java s'estructura en projectes, que contenen classes. Cada classe té una sèrie de mètodes i variables entre d'altres, que en dicten el funcionament. Cada un d'aquests arxius s'anomenen amb l'extensió *.java*. Després de la compilació, obtindrem els arxius *.class* corresponents, que ja poden ser executats per la màquina virtual de Java (JVM).

Per a programar qualsevol aplicació, primer de tot, ens haurem d'instal·lar un equip de desenvolupament en Java (JDK), que conté, entre d'altres, el compilador i les biblioteques de classes d'utilitat general. No ho hem de confondre amb el JRE (*Java Runtime Environment*), que inclou components necessaris per a l'execució de programes en Java.

Normalment, per a programar aplicacions en el llenguatge de programació Java, s'utilitza un IDE, això és, una interfície gràfica que ens facilita la programació, la detecció d'errors, la compilació i l'execució de les aplicacions. N'hi ha moltes, però probablement *Eclipse* i *Netbeans* són les més populars.

Java es basa en una **programació estructurada**. Normalment tindrem una classe que coincidirà amb el nom del fitxer que contingui el codi del programa. L'execució del codi s'ha d'iniciar en el mètode `main()` d'aquesta classe. Aquesta funció es considera el punt d'entrada de l'aplicació, on s'inicia el pro-

cés. En conseqüència, les instruccions que contingui el mètode `main()` s'han d'executar en ordre seqüencial i seguint els salts de codi, segons hagi indicat el programador.

Vegem ara un esquelet bàsic del que seria un programa escrit en aquest llenguatge de programació. En aquest cas, hauria de guardar-lo en un arxiu anomenat *prova.java*.

```
import java.io.*; //llibreries
public class prova {  //classe prova
    public static void main(String argv[]) {
        //primer mètode que s'executarà
        //codi del programa
    }
}
```

4.2. Excepcions en Java

Java té el seu propi sistema de gestió d'excepcions o errors, basat en un mecanisme de programació per esdeveniments.

Un procés va executant les instruccions de codi segons dicta el programa que executa. Si el programador no gestiona les excepcions i en qualsevol d'aquestes instruccions es produeix un error, el procés finalitza per l'error inesperat. En canvi, si el programador gestiona les excepcions, el procés en continua la seva execució i es redirigeix cap a un seguit d'instruccions, segons s'hagi programat.

La gestió d'excepcions és molt habitual i necessària quan programem aplicacions en xarxa, ja que els errors són comuns i no han d'implicar que el programa aborti de manera inesperada. Per exemple, imaginem que un servidor hagi caigut o que no ens pugui atendre perquè hi ha sobrecàrrega de clients. El client programat podria buscar un servidor alternatiu o esperar a que la situació es resolgui i tornar-ho a intentar, comunicant-ho a l'usuari amb un missatge informatiu. Un bon programador no només ha d'escriure les instruccions perquè el procés s'executi quan no hi ha errors; ans al contrari, ha de preveure totes les possibilitats i plantejar camins alternatius de codi, quan les funcions executades no responguin segons s'espera.

Les paraules reservades per a capturar i tractar excepcions en Java són:

- ***Try***: dins d'aquest bloc hi posarem el codi del programa que volguem sotmetre a «prova», és a dir, es capturaran les excepcions del codi que estigui dins el bloc `try`.

- ***Catch***: aquest bloc recull les intruccions de codi que s'executaran quan es produeixi un error en alguna part del bloc anterior, això és, conté el tractament d'errors.

- ***Finally***: aquesta part conté un bloc d'instruccions de codi que s'executen sempre, tant si hi ha error com si no.

Veiem-ne un exemple senzill. Abans de continuar, intenteu entendre el codi i penseu quin seria el resultat d'aquest programa.

```
import java.io.*;
public class prova {
    public static void main (String [] args) {
        try {
            System.out.println("Prova1”);
            int num = Integer.parseInt("E");
            System.out.println("Prova2”);
        } catch (Exception e) {
            System.out.println("Prova3");
        } finally {
            System.out.println("Prova 4");
        }
    }
}
```

Fixeu-vos que la línia de codi referent a la funció `parseInt()` produeix error, perquè li estem passant com a paràmetre un caràcter quan espera un enter. El resultat d'executar el codi anterior seria:

```
Prova1
Prova3
Prova 4
```

4.3. Les classes Java sobre *sockets*

Després d'aquestes pinzellades bàsiques de Java, ens endinsem en la programació d'aplicacions en xarxa mitjançant *sockets*. Dins el paquet *java.net*, Java proporciona les següents classes per a facilitar la programació en *sockets*, segons la naturalesa del mateix:

- ***DatagramSocket***: Classe encarregada de dur a terme comunicacions no fiables, no orientades a la connexió (protocol UDP). La unitat d'enviament és un datagrama, simbolitzat en una instància de l'objecte *DatagramPacket*.

- ***Socket***: Objecte bàsic en una comunicació per xarxa. Representa un *socket*, això és, un dels extrems de la connexió o del *socket pair*, en comunicacions fiables orientades a la connexió (que usen TCP com a protocol de

transport). Els clients es connectaran a un determinat servidor mitjançant aquest objecte *Socket*. També, tant clients com servidors escriuran i llegiran les dades que s'intercanviïn.

- ***ServerSocket***: Classe encarregada d'implementar el Servidor de la connexió en comunicacions fiables orientades a la connexió (protocol TCP). Representa el *socket* d'escolta, mitjançant el qual el servidor està a l'espera de peticions de connexió de potencials clients. En acceptar la petició de connexió d'un client, retorna una instància de la classe *Socket*, per on realment es durà a terme la comunicació amb el client.

- ***SocketImpl***: Classe abstracta per a poder definir qualsevol tipus de comunicació, és a dir, per a poder configurar els *sockets* sense haver-nos de cenyir a les propietats estàndards dels *sockets* TCP o UDP descrits anteriorment. Si instanciem una subclasse de *SocketImpl*, podem redefinir-ne els seus serveis per tenir un control ple de la comunicació. Per exemple, si volguéssim implementar un *firewall*, hauríem de redefinir el mètode `accept()` afegint els controls de seguretat necessaris. De fet, totes les classes enumerades són instàncies de la classe *SocketImpl*.

Una altra classe emprada sovint a l'hora de programar aplicacions en xarxa és:

- ***InetAddress***: Classe encarregada d'implementar una adreça IP.

4.4. Programació de *sockets* no orientats a la connexió

Com ja hem vist en l'apartat «*Sockets* UDP o no orientats a la connexió», UDP és un protocol no fiable que permet l'enviament de datagrames a través d'una xarxa sense que s'hagi establert prèviament una connexió, ja que la capçalera del datagrama mateix que s'intercanvien els extrems de la comunicació incorpora informació d'adreçament com per a poder saber el remitent i el destinatari.

La seqüència d'instruccions de programació típica d'un **servidor seqüencial** que atèn múltiples clients és la següent:

- Crear un *socket* no orientat a la connexió que estigui a l'espera de peticions de clients en el port configurat.
- Llegir el datagrama relatiu a la petició que ens envia un client a través d'aquest *socket* d'escolta.
- Respondre aquesta petició, escrivint el datagrama de resposta pel mateix *socket*.
- Repetir els passos 2 i 3 tantes vegades com sigui necessari.
- Tancar el *socket* per alliberar els recursos de sistema.

En comunicacions no orientades a la connexió, recordem que els datagrames que s'intercanvien clients i servidors contenen el *socket pair*, això és, les adreces IP i ports dels dos extrems de la comunicació.

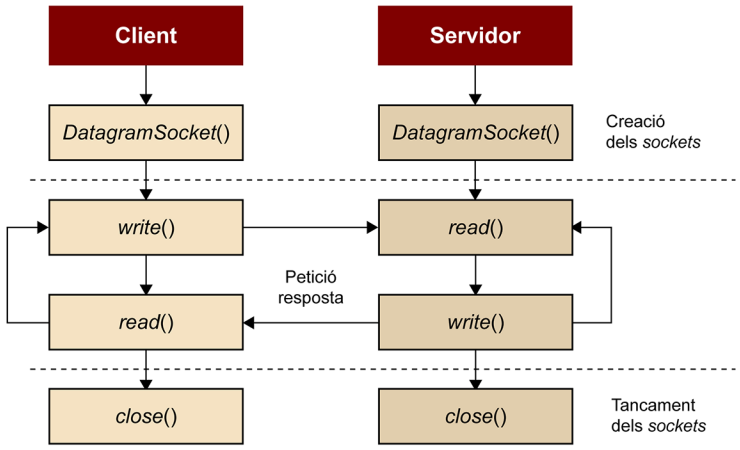
Un altre punt a ressaltar en aquest tipus de comunicacions és que el servidor pot rebre peticions de qualsevol client. En no haver-hi un establiment de la comunicació ni un canal bidireccional únic entre servidor i un client, només hi ha un *socket* per on es reben les peticions i es responen. Aquest fet implica que podem intercalar peticions de diversos clients alhora, ja que no existeix una connexió dedicada.

D'altra banda, el **client** ha d'emprar un mecanisme similar, però és la part activa de la comunicació:

- Crear un *socket* no orientat a la connexió per on comunicar-se amb el servidor.
- Enviar-li la petició de servei al servidor, escrivint el datagrama al *socket* creat, indicant el *socket* (l'adreça IP i port) del servidor a qui va dirigida.
- Llegir el datagrama relatiu a la resposta que ens envïi el servidor.
- Repetir els passos 2 i 3 tantes vegades com sigui necessari.
- Tancar el *socket* per alliberar els recursos de sistema.

En comunicacions no orientades a la connexió, tant client com servidor utilitzen les mateixes classes amb els seus mètodes, per la qual cosa les explicarem genèricament i després veurem un exemple concret per a cada un d'aquests dos rols. El gràfic genèric dels mètodes implicats en el procés anterior és:

Figura 11.



4.4.1. Crear un *socket*

Per a crear un nou *socket* UDP, bé sigui servidor o client, el primer que hem de fer és crear un nou objecte de la classe `DatagramSocket()`, per on enviar i rebre datagrames, utilitzant un dels següents constructors:

- `DatagramSocket()`: crea un *socket* UDP, assignant-li un número de port disponible de la màquina local.
- `DatagramSocket(int port)`: crea un *socket* i l'assigna al port passat com a paràmetre.
- `DatagramSocket(SocketAddress binaddr)`: crea un *socket* UDP, assignant-li l'adreça passada com a paràmetre.
- `DatagramSocket(int port, InetAddress laddr)`: crea un *socket* i l'assigna a l'adreça i port passats com a paràmetre.

Les excepcions que poden produir-se arrel de l'execució d'aquestes crides són:

- `SocketException`: si el *socket* no es pot obrir.
- `SecurityException`: si el gestor de seguretat del dispositiu no permet crear el *socket*.

Normalment, en el cas dels clients, utilitzem el primer dels constructors enumerats, deixem que el sistema sigui qui elegeixi crear el *socket* amb l'adreça IP de la màquina i un dels ports lliures. En cas dels servidors, és habitual utilitzar el constructor descrit com a segon punt de la enumeració, ja que es necessita crear un *socket* amb nom conegut i, per tant, necessitem conèixer *a priori* el port per on estarà oferint serveis per a que els clients li puguin demanar. Per exemple, el següent servidor no orientat a la connexió escolta peticions de clients pel port 6000.

```
try {
    DatagramSocket se = new DatagramSocket (6000);
} catch (SocketException ex) {
    System.err.println(ex);
}
```

4.4.2. El datagrama

En el transcurs de la comunicació entre clients i servidors no orientats a la connexió, la unitat del paquet enviat és el datagrama. En Java, la classe que representa el datagrama és `DatagramPacket()`. Com hem comentat, a part de les dades en si, ha de contenir l'adreça del destinatari per a poder adreçar-hi el paquet. Els constructors més rellevants d'aquesta classe són:

- `DatagramPacket(byte[] buf, int len, InetAddress address, int port)`
- `DatagramPacket(byte[] buf, int len, SocketAddress address)`

En els dos casos, es construeix un paquet de tipus datagrama, que conté:

- Les dades passades com a primer paràmetre, de longitud `len`.
- El *socket* destinatari identificat per una adreça IP i port.

Donat un objecte de tipus datagrama sempre podem consultar o configurar els atributs anteriors, amb els mètodes següents:

- `InetAddress getAddress()`
- `byte[] getData()`
- `int getLength()`
- `int getPort()`
- `SocketAddress getSocketAddress()`
- `setAddress(InetAddress iaddr)`
- `setData(byte[] buf)`
- `setLength(int length)`
- `setPort(int iport)`
- `setSocketAddress(SocketAddress address)`

4.4.3. Enviar dades

Com hem vist, les comunicacions orientades a la connexió estan basades en un protocol petició-resposta. Tant per a enviar la petició del client al servidor, com per a enviar la resposta del servidor al client, la crida per excel·lència de la classe `DatagramSocket()` és `send()`:

- `send(DatagramPacket p)` : envia un datagrama pel *socket*. Aquest datagrama passat com a paràmetre conté les dades que es volen enviar, l'adreça IP i el port del destinatari, tal com s'ha especificat en l'apartat anterior.

El mètode `send()` pot provocar les següents excepcions principalment:

- `IOException`: si ocorre un error d'entrada sortida.
- `SecurityException`: si el gestor de seguretat del dispositiu no permet l'enviament de datagrames pel *socket*.
- `PortUnreachableException`: si no es pot trobar el *socket* destí. No hi ha certesa que aquesta excepció sempre es llanci.

El següent exemple construeix un datagrama que porta com a destinatari un servidor local que s'està executant al port 6000 i li envia un missatge.

```
import java.net.*;
import java.io.*;
public class clienttcp {
    public static void main(String argv[]) {
        InetAddress adr;
        String missatge = "";
        byte[] missatge_bytes = new byte[256];
        DatagramPacket paquet;
        try {
            DatagramSocket socket = new DatagramSocket (6001);
            adr = InetAddress.getByName("localhost");
```

```
            missatge = "Petició a enviar al servidor: ";
            missatge_bytes = missatge.getBytes();
            paquet = new DatagramPacket(missatge_bytes, missatge.length(), adr, 6000);
            socket.send(paquet);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

4.4.4. Llegir dades

La lectura dels datagrames que ens arriben per un *socket* no orientat a la connexió pot procedir, o bé de les peticions dels clients en el servidor, o bé de les respostes dels servidors en els clients. El mètode per llegir del *socket* en qualsevol dels dos casos és `receive()`:

- `receive(DatagramPacket p)`: llegir un datagrama pel *socket*. Aquest datagrama contindrà dades, l'adreça IP i el port del remitent, segons s'ha especificat en el subapartat 4.4.2.

Podem destacar les següents excepcions d'aquest mètode:

- `IOException`: si ocorre un error d'entrada sortida.
- `PortUnreachableException`: si no es pot trobar el *socket* destí. No hi ha certesa que aquesta excepció sempre es llanci.
- `SocketTimeoutException`: si s'esgota el temps d'espera que s'ha configurat per a la recepció del datagrama.

El mètode `receive()` és bloquejant, és a dir, el procés espera en aquesta instrucció fins que es rebi un datagrama. Si es vol configurar aquest temps d'espera, podem emprar les següents funcions:

- `setSoTimeout(int timeout)`: activa o desactiva SO_TIMEOUT amb el temps especificat com a paràmetre, en mil·lisegons. Si és 0, la crida `receive` serà bloquejant fins que es rebi un datagrama.
- `int getSoTimeout()`: per a obtenir el valor actual de SO_TIMEOUT.

En cas d'esgotar-se el temps d'espera mentre el procés està esperant en la crida `receive()`, es llança l'excepció *SocketTimeoutException*, que podrem tractar amb les instruccions que es creguin convenients, en la secció de tractament d'errors del codi.

A continuació, veurem un senzill exemple de lectura d'un datagrama. Un cop rebut, se'n mostra per pantalla el seu contingut.

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

```
import java.io.*;
public class provaudp {
    public static void main(String argv[]) {
        try {
            DatagramSocket socket= new DatagramSocket (6000);
            byte[] missatge_bytes = new byte[256];
            DatagramPacket paquet = new DatagramPacket(missatge_bytes, 256);
            socket.receive(paquet);
            System.out.println("Adreça IP del remitent: " + paquet.getAddress());
            System.out.println("Port del remitent: " + paquet.getPort());
            String missatge = new String(paquet.getData());
            System.out.println("Dades rebudes: " + missatge);
            System.out.println("Longitud de les dades rebudes: " +
            paquet.getLength());
            socket.close();
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

4.4.5. Tancar una connexió

Per a tancar un *socket*, s'ha d'invocar al mètode `close()` sobre la instància de *socket* que ens comunica explícitament amb el client:

- `close()`: tanca un *socket*.

La excepció que pot llançar aquesta crida és:

- `IOException`: si ocorre un error d'entrada sortida.

Un exemple seria:

```
sc.close();
```

4.4.6. Exemple d'un client i un servidor

A continuació, es mostra un exemple complet i funcional d'un client i un servidor no orientats a la connexió. El funcionament general és el següent:

- El client envia al servidor tot el que l'usuari li introdueix per teclat, fins que rep la paraula *end*.
- El servidor llegeix tot el que li envien els clients i ho mostra per pantalla.

És aconsellable que copieu el codi del servidor en un arxiu anomenat *servidorudp.java* i el codi del client en un arxiu anomenat *clienteudp.java*. Els podeu compilar i executar per provar la interacció entre client i servidor. La seva execució seria similar a la següent pantalla:

```
escrivint
al
servidor
end
END
```

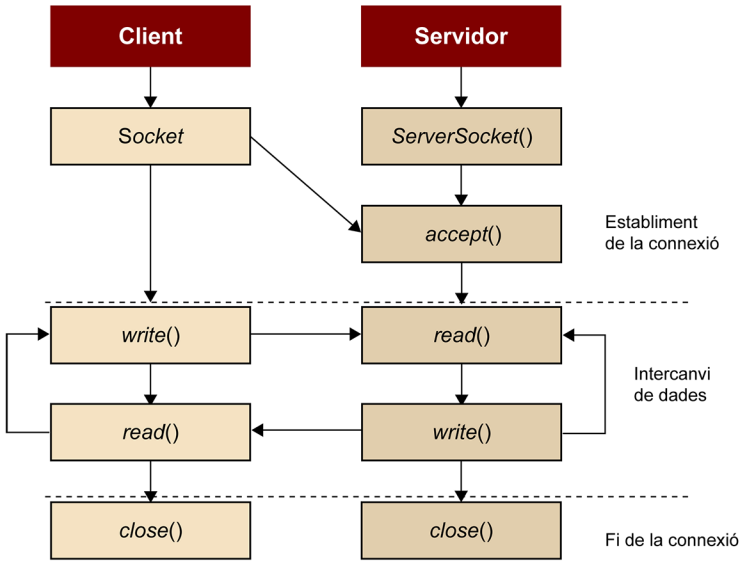
```
/** SERVIDOR NO ORIENTAT A LA CONNEXIÓ **/
/** Llibreries habituals quan treballem amb sockets **/
import java.net.*;
import java.io.*;
/** Es crea una nova classe Java anomenada Servidor UDP i una nova funció main que en governa
el funcionament. En primer lloc es declaren dues variables, una de tipus socket Servidor UDP
i una altra com a missatge per a gestionar les dades que intercanviarem amb els potencials clients **/
public class servidorudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        String missatge = "";
/** Es crea un socket servidor en UDP anomenat socket pel port 6000 que espera connexions
    de potencials clients. **/
        try {
            socket = new DatagramSocket(6000);
/** Es declaren i s'inicialitzen les variables que llegiran les dades que ens envia el client
    mitjançant el socket que s'hi ha establert. **/
            byte[] missatge_bytes = new byte[256];
            DatagramPacket paquet = new DatagramPacket(missatge_bytes, 256);
/*S'inicia l'intercanvi de missatges mitjançant un bucle que llegeix el que el client envia i
    ho mostra per pantalla fins que el client envia la paraula "END" **/
            do {
                socket.receive(paquet);
                missatge = new String(missatge_bytes);
                System.out.println(missatge);
            } while (!missatge.startsWith("END"));
/** Tanquem el socket que ens permetia comunicar-nos amb el client **/
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
}
```

```
/** CLIENT NO ORIENTAT A LA CONNEXIÓ **/
/** Llibreries habituals quan treballem amb sockets **/
import java.net.*;
import java.io.*;
/** Es crea una nova classe Java anomenada Client UDP i una nova funció main que en governa el funcionament. I
public class clientudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        InetAddress adr;
        String missatge = "";
        byte[] missatge_bytes = new byte[256];
        DatagramPacket paquet;
        BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
        missatge_bytes = missatge.getBytes();
/** Es crea un socket client anomenat socket que es connectarà al servidor UDP. També en la
    variable adr es posa l'adreça del servidor que hem passat com a paràmetre del programa **/
        try {
            socket = new DatagramSocket();
            adr = InetAddress.getByName(argv[0]);
/** S'inicia l'intercanvi de missatges mitjançant un bucle que llegeix el que l'usuari introdueix
    per teclat i li envia al servidor, fins que escrivim la paraula "END" **/
            do {
                missatge = entrada.readLine();
                missatge_bytes = missatge.getBytes();
                paquet = new DatagramPacket(missatge_bytes, missatge.length(), adr, 6000);
                socket.send(paquet);
            } while (!missatge.startsWith("end"));
/** Tanquem el socket client que ens permetia comunicar-nos amb el servidor **/
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
}
```

5. Programació de *sockets* orientats a la connexió

TCP és un protocol orientat a la connexió, que permet el lliurament de segments de manera fiable, tal com s'ha explicat en el subapartat 3.2. Abans de poder transmetre cap dada, és necessari establir una connexió entre els dos extrems que es volen comunicar, per on s'intercanvien els missatges. A continuació, veurem el diagrama de flux genèric de client i servidor.

Figura 12.



5.1. El servidor TCP

En comunicacions orientades a la connexió, la seqüència d'operacions habitual que un **servidor seqüencial** realitza en el transcurs de la comunicació amb els clients és la següent:

- Crear un *socket* servidor i assignar-li un determinat número de port conegut perquè els potencials clients s'hi puguin connectar.
- Escoltar noves connexions de clients i acceptar-les. Com a resultat, es crea un nou *socket* que comunicarà client i servidor. El *socket* d'escolta es mantindrà a l'espera de noves connexions.
- Llegir dades que ens envia el client, mitjançant el *socket pair* establert amb ell.
- Enviar dades al client mitjançant el *socket pair*.
- Repetir els passos 3 i 4 tantes vegades com sigui necessari.
- Tancar la connexió amb el client.
- Repetir els passos 2 en endavant.

A continuació veurem la classe de Java *ServerSocket* del paquet *java.net* en detall, per tal de veure els mètodes que implementen aquests passos d'un servidor seqüencial típic. L'intercanvi de dades entre client i servidor, el veurem més endavant, ja que les crides són les mateixes pels dos extrems de la comunicació.

5.1.1. Crear un *socket*

Per a crear un nou *socket* servidor que es posi a escoltar noves peticions de connexions per part dels clients, el primer que hem de fer és crear un nou objecte de la classe `ServerSocket`, utilitzant un dels següents constructors:

- `ServerSocket(int port)`: crea un *socket* servidor vinculat al port passat com a paràmetre. Si el port és 0, s'assigna un port aleatori, normalment un número dins del rang de ports efímers. El nombre màxim de clients que es volen connectar al servidor i estan esperant a la cua és 50. A partir d'aquest nombre, els nous clients es descarten.

- `ServerSocket(int port, int backlog)`: crea un *socket* servidor vinculat al port passat com a paràmetre. El màxim nombre de connexions a la cua és l'especificat al paràmetre `backlog`.

- `ServerSocket(int port, int backlog, InetAddress bindAddr)`: crea un *socket* servidor vinculat al port i l'adreça IP passats com a paràmetre, amb el nombre màxim de connexions a la cua que especifica el paràmetre `backlog`. Especificar l'adreça IP és útil quan tenim varies interfícies de xarxa a la màquina on s'executa el servidor: wifi, ethernet... També es pot emprar quan volem que el servidor només accepti clients d'una determinada adreça.

De les opcions enumerades, la més utilitzada és la primera, per la seva facilitat d'ús. Fixeu-vos que, com ens trobem en el servidor, sempre hem d'indicar un número de port, que ha de ser conegut per la resta de clients potencials que s'hi vulguin connectar. Sense l'adreça IP i el port d'aquest servidor, que són els que identifiquen el *socket*, seria impossible que els clients sàpiguen on han de connectar-se.

Les crides anteriors poden llançar les següents excepcions:

- `IOException`: si ocorre algun error quan obrim el *socket*.

- `SecurityException`: si existeix un gestor de seguretat, com uns tallafocs, que no permet l'operació realitzada.

- `IllegalArgumentException`: si els paràmetres no són correctes. Per exemple, si posem un número de port que està fora del rang de ports vàlids, de 0 a 65535 inclòs.

Per exemple, el següent codi crea un nou *socket* servidor a la variable `se`, i mostra per pantalla un missatge en cas d'error :

```
try {
    ServerSocket se = new ServerSocket(80);
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.1.2. Acceptar una connexió

Un cop hem creat una instància d'objecte `ServerSocket`, el següent pas és començar a escoltar peticions de nous clients que es volen connectar amb el servidor i acceptar-les, emprant el mètode `accept()`. Aquest mètode bloqueja el procés en curs fins que es fa una connexió amb el client. És a dir, no passarem a la següent instrucció de codi fins que un nou client es connecti al servidor.

- `Socket accept()`: es manté el *socket* a l'espera d'una nova petició d'un client; quan aquesta arriba, l'accepta. Retorna el *socket* creat, que formarà part del *socket pair* que el vincularà amb l'extrem client i per on es vehicularà la comunicació bidireccional.

Les crides anteriors poden llançar les següents excepcions:

- `IOException`: si ocorre algun error quan estem esperant una nova connexió d'un client.

- `SecurityException`: si existeix un gestor de seguretat, com uns tallafocs, que no permet l'operació realitzada.

- `IllegalBlockingModeException`: si un *socket* té un canal associat, el canal es troba en mode no bloquejant i no hi ha una connexió preparada ja per ser acceptada.

Continuant l'exemple de l'apartat anterior, tindríem:

```
try {
    ServerSocket se = new ServerSocket(80);
    Socket sc = se.accept();
} catch (IOException ex) {
    System.err.println(ex);
}
```

Fixeu-vos que el mètode `accept` retorna un nou objecte `socket (sc)` que s'ha creat per a comunicar-se explícitament amb el client, del qual se li ha acceptat la petició. És en aquest on llegirem les peticions dels clients i pel que enviarem les respostes del servidor. Així doncs, el `socket sc` del servidor, juntament amb el *socket* creat en la part client formen el *socket pair* mitjançant el qual es produeix la comunicació bidireccional entre client i servidor. En canvi, el `socket se` és un *socket* que es manté en mode espera a l'escolta de noves peticions de clients.

D'altra banda, hi ha la possibilitat de limitar el temps durant el qual el mètode `accept()` estigui bloquejat esperant un nou client:

- `setSoTimeout(int timeout)`: configura el temps en mil·lisegons que el servidor està esperant noves connexions de clients, és a dir, el temps que el procés està bloquejat a la crida `accept()`. Quan passi el temps especificat, es llança una excepció `SocketTimeoutException`, que haurem de tractar segons convingui.

- `int getSoTimeout()`: relacionat amb la crida anterior, retorna el temps configurat a SO_TIMEOUT.

5.1.3. Tancar una connexió

Per a tancar un *socket*, s'ha d'invocar al mètode `close()` sobre la instància de *socket* que ens comunica explícitament amb el client o que es troba escoltant peticions de nous clients:

- `close()`: tanca un *socket*.

La crida anterior pot llançar l'excepció:

- `IOException`: si ocorre un error d'entrada o sortida quan es realitza el tancament del *socket*.

Quan haguem finalitzat la comunicació amb un determinat client, podem tancar el *socket* que era extrem local del servidor en el *socket pair*.

D'altra banda, un servidor sempre ha d'estar actiu, esperant connexions de nous clients als quals servir. En el cas que el vulguem parar per qualsevol motiu, podem emprar el mètode *close*() anterior sobre el *socket* d'escolta.

Aquesta instrucció és especialment rellevant a l'hora de programar i fer proves, ja que, en cas contrari, es deixa el procés en un estat inconsistent i ocorren problemes per a reusar la mateixa connexió (mateix número de port en el mateix equip).

El següent codi, accepta només la connexió d'un client i finalitza tancant els *sockets* creats.

```
try {
    ServerSocket se = new ServerSocket(8000);
    Socket sc = se.accept();
    sc.close();
    se.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.2. El client TCP

Ara que ja hem vist les principals classes i els seus mètodes en l'extrem del servidor, veurem en detall la part del client. La seqüència d'operacions és la següent:

- Crear un *socket* client, indicant-li l'adreça IP i el port del servidor al qual es vol connectar.
- Enviar dades al servidor, mitjançant el *socket pair* que s'hi ha establert.
- Llegir dades que ens envia el servidor, mitjançant el *socket pair* establert.
- Repetir els passos 2 i 3 tantes vegades com es vulgui.
- Tancar la connexió amb el servidor.

A continuació vegem la classe *Socket* del paquet *java.net* en detall, que és la classe que implementa un *socket* client i, recordeu, també el *socket* servidor que es comunica amb el client, de manera que formen el *socket pair*.

5.2.1. Establir una connexió

Per a connectar-se a un servidor, hem de crear un nou objecte `Socket` emprant un dels següents constructors:

- `Socket(InetAddress address, int port)`: crea un *socket* client que es connecta al servidor que té l'adreça IP i el port indicats com a paràmetres.

- `Socket(String host, int port)`: crea un *socket* client que es connecta al servidor, que té el nom de domini i el port indicats com a paràmetres.

- `Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`: crea un *socket* client a partir dels dos últims paràmetres, que es connecta al servidor indicat als dos primers paràmetres.

- `Socket(String host, int port, InetAddress interface, int localPort)`: el mateix que l'anterior, però hi indiquem explícitament la interfície de xarxa sobre la qual treballarem. S'empra en equips amb dos o més connexions a Internet, per exemple Wifi i Ethernet.

L'opció més utilitzada és la segona, atesa la seva facilitat d'ús. Per exemple, la següent línia de codi intenta connectar-se al port 80 del domini *uoc.edu*, creant un nou *socket* client a la variable *c*:

```
Socket c = new Socket("uoc.edu", 80);
```

El port del client l'assigna automàticament el sistema operatiu dins del rang de ports efímers (de 49151 a 65535). L'aplicació per la banda del client assigna el número de port de manera automàtica i transparent, mentre que a la part servidora el *socket* d'escolta ha de ser conegut.

Aquests constructors poden llançar les excepcions següents, principalment:

- `IOException`: si ocorre algun error d'entrada o sortida quan creem el *socket*.

- `UnknownHostException`: si l'adreça IP del dispositiu no es pot resoldre (en crides on passem un nom de domini).

- `SecurityException`: si existeix un gestor de seguretat, com uns tallafocs, que no permet l'operació realitzada.

- `IllegalArgumentException`: si els paràmetres no són correctes. Per exemple, si posem un número de port que està fora del rang de ports vàlids, de 0 a 65535 inclòs.

Tal com hem vist anterior, podem gestionar aquestes excepcions quan creem una instància d'un *socket*:

```
try {
    Socket c = new Socket("uoc.edu", 80);
} catch(IOException e1){
    e1.printStackTrace();
    System.err.println(e1);
}
```

5.2.2. Tancar una connexió

Per a tancar un *socket*, s'ha d'invocar al mètode `close()` sobre la instància de *socket* que ens comunica explícitament amb el client. El funcionament és el mateix que ja vam veure en el servidor:

- `close()`: tanca un *socket*.

La crida anterior pot llançar l'excepció:

- `IOException`: si ocorre un error d'entrada o sortida quan es realitza el tancament del *socket*.

El següent codi és un exemple molt senzill d'un client que es connecta a un servidor i tanca la connexió:

```
try {
    Socket c = new Socket("uoc.edu", 80);
    c.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.3. Els fluxos de comunicació en TCP

Quan estem programant *sockets* en Java, a part d'utilitzar les classes ofertes pel *packet java.net*, és habitual treballar conjuntament amb el paquet *java.io*, que conté un conjunt de canals d'entrada i sortida que podem utilitzar per a llegir i escriure dades fàcilment. Concretament, s'utilitzen abstraccions de les operacions de lectura i escriptura dels anomenats fluxos de dades o *streams*. Aquesta abstracció permet usar el mateix model amb independència del tipus de dades intercanviades i del dispositiu que du a terme aquestes operacions.

És a dir, s'utilitzen les mateixes classes i mètodes per a gestionar fitxers, la pantalla d'un dispositiu o els *sockets*. Aquest fet ofereix una gran flexibilitat en la programació.

> Un flux o *stream* és una seqüència de dades que s'intercanvien entre un origen i un destí de la comunicació. Java defineix principalment dos tipus de fluxos: fluxos de *bytes* (*byte streams*) i fluxos de caràcters (*character streams*).

Els **fluxos o *byte streams*** gestionen canals d'entrada i sortida de bytes, per exemple, en llegir i escriure dades binàries, i són especialment útils quan treballem amb arxius. Les següents classes treballen sobre fluxos de bytes (*byte streams*):

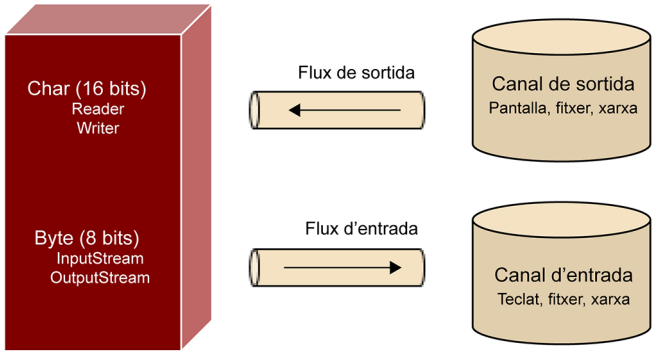
- *InputStream*: classe per a la lectura.
- *OutputStream*: classe per a l'escriptura.

Un altre tipus són els **fluxos o *streams* de caràcters**, per a gestionar l'entrada i sortida de text Unicode i, per tant, poden ser internacionalitzats:

- *Reader*: classe per a la lectura.
- *Writer*: classe per a l'escriptura.

Els fluxos de *bytes* i caràcters **es poden combinar** sobre el canal d'entrada i sortida en qüestió segons ens convingui, i es poden emprar els dos alhora, amb la instància corresponent. Per exemple, podem tenir un protocol de comunicacions que utilitzi una capçalera com a conjunt de caràcters i el contingut d'un fitxer binari. En tal cas, utilitzaríem un flux de caràcters per a la capçalera i un flux de tipus binari per al fitxer. Podríem intercanviar-nos dades emprant els mètodes que cada tipus de flux ens ofereix, emprant el mateix canal.

Figura 13.

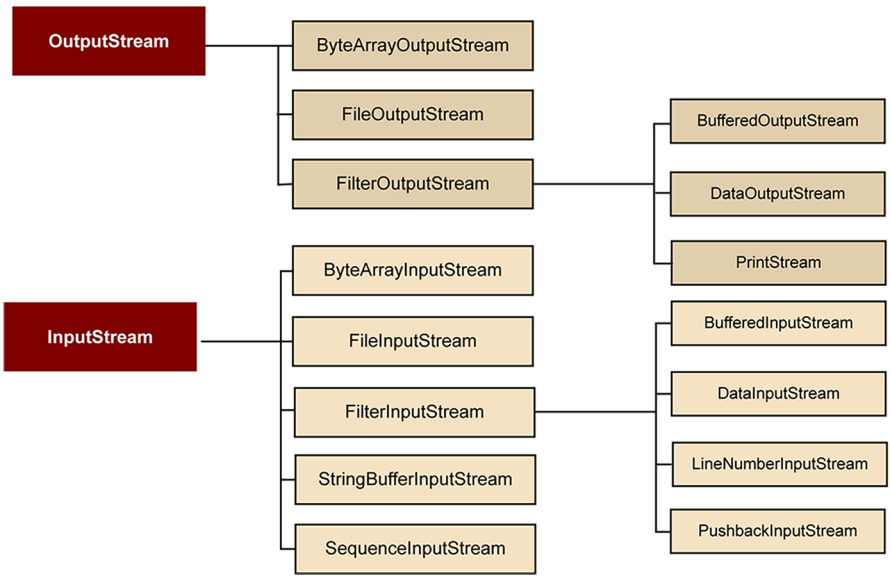


També es pot **traduir** o passar d'un flux de bytes a un de caràcters Unicode, codificant les dades, i a la inversa, utilitzant les següents classes:

- *InputStreamReader*: llegeix bytes i els decodifica com a caràcters.
- *OutputStreamWriter*: rep caràcters i els codifica en bytes.

Tant les classes exposades per a treballar amb un flux de tipus binari com les que treballen amb un flux de caràcters, no es poden usar directament, ja que són classes abstractes. Aquestes classes hauran de ser esteses per altres sobre les que sí podem treballar i, així doncs, fer les operacions d'entrada i sortida que correspongui sobre objectes. Aquest **recobriment o *wrapper*** és habitual en Java, i proporciona un conjunt de classes que ofereixen mètodes per a la manipulació dels objectes de les classes abstractes. La jerarquia de subclasses de *InputStream* i *OutputStream* que implementen tipus específics de canals d'entrada i sortida és extensa, així que només en destacarem les següents, per ser d'ús popular quan programem *sockets*.

Figura 14.



Si són **fluxos binaris** és habitual treballar llegint i escrivint dades, mitjançant les classes:

- *DataInputStream*: per al canal d'entrada (lectura).
- *DataOutputStream*: per al canal de sortida (escriptura).

Si estem treballant amb **fluxos de caràcters**, no existeix aquesta opció directament, però sí la classe *PrintWriter* per a escriure dades formatades.

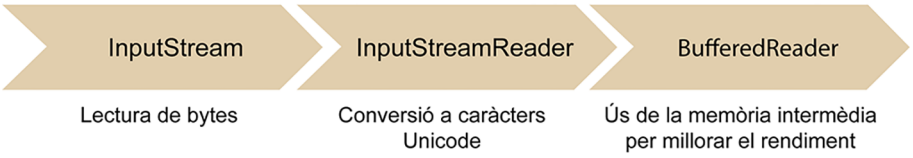
També és habitual emprar la tècnica dels ***buffers*** per la seva eficiència a l'hora de llegir i escriure. Utiltitzen un *buffer* intern on es van emmagatzemant els caràcters i així podem triar el ritme de lectura si no és constant, optimitzant el rendiment del procés. Les classes que doten d'aquesta funcionalitat són:

- *BufferedReader*: per al canal d'entrada (lectura).

- *BufferedWriter*: per al canal de sortida (escriptura).

A continuació, treballarem aquests conceptes amb un exemple pràctic: el teclat. La classe Java *System* representa el canal d'entrada i sortida estàndar. El teclat es representa com a *System.in*, que és de tipus *InputStream*, això és, flux de bytes. Com normalment al treballar amb el teclat, utilitzem caràcters i no bytes, crearem un objecte *InputStreamReader* a partir de *System.in*. Ara que ja tenim el canal d'entrada, el podem recobrir amb un altre per a dotar-lo de les funcionalitats que ens convingui, com, per exemple, el *BufferedReader*. ´

Figura 15.



Un cop definida la classe que representa el canal d'entrada o sortida de bytes o caràcters, ja es poden emprar les funcions habituals per a llegir i escriure dades, basades en `read()` i `write()`. Cada classe ofereix un conjunt de funcions pròpies, que difereixen lleugerament entre sí, segons els matisos que introdueixen. Per exemple, llegir o escriure bytes, caràcters, línies, etc.

```
import java.io.*;
public class eco {
    public static void main (String[] args){
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String msg;
        System.out.println("Introdueix una frase: ");
        msg = br.readLine();
        System.out.println("La frase és: " + msg);
    }
}
```

5.3.1. Enviar dades

El mecanisme de lectura i escriptura per al client i servidor són idèntics. Una vegada s'ha establert el *socket* entre client i servidor, podem gestionar els fluxos de bytes utilitzant les superclasses `InputStream` i `OutputStream`.

Vegem primer l'escriptura. Per a enviar dades binàries a baix nivell (array de bytes), s'empra una instància de la classe `OutputStream`. Les funcions principals són les següents:

- `OutputStream()`: constructor de la classe.
- `write(int b)`: escriu el byte indicat al flux de sortida (output stream).
- `write(byte[] b)`: escriu tots els bytes passats com a paràmetre.

- `write(byte[] b, int off, int len)`: escriu *len* bytes del paràmetre b, començant des de la posició *off* (offset). El primer element és *b[off]* i l'útim *b[off+len-1]*.
- `flush()`: força l'escriptura de qualsevol byte pendent al *buffer*, és a dir, immeditament s'escriu al flux de sortida tot el que s'havia indicat.
- `close()`: tanca el flux de sortida i allibera els recursos de sistema associats.

L'excepció més rellevant és:

- `IOException`: si ocorre qualsevol error d'entrada sortida.

Un exemple de codi seria:

```
OutputStream sortida = socket.getOutputStream();
byte[] dades = {0x5b, 0x42, 0x40, 0x34};
sortida.write(dades);
```

Si volem enviar dades en format text podem convertir el flux de sortida a altres classes, que hereten el comportament essencial però afegeixen funcionalitats particulars. Una d'aquestes classes envolvents és `DataOutputStream`, la qual permet escriure tipus de dades primitius de Java. A part dels mètodes explicats abans, podem destacar els següents:

- `DataOutputStream(OutputStream out)`: constructor de la classe.
- `writeByte(int v)`: escriu un byte al flux de sortida.
- `writeChar(int v)`: escriu un caràcter al flux de sortida, això és, dos bytes, el byte de més pes primer.
- `writeBytes(String s)`: escriu una seqüència de bytes, caràcter a caràcter, com a un sol byte.
- `writeChars(String s)`: és el mateix que l'anterior, però cada caràcter s'escriu com a dos bytes.
- `writeUTF(String s)`: escriu la cadena de caràcters utilitzant la codificació UTF-8, creant independència de la màquina.
- `writeInt(int v)`: escriu un enter al flux de sortida, això és, quatre bytes, el byte de més pes primer.
- `writeLong`, `writeFloat`, `writeDouble`, ...

Com veiem, emprant la classe **`DataOutputStream`**, la riquesa de mètodes oferts és més gran, cosa que facilita la tasca del programador. Un exemple per a escriure un caràcter o una cadena de caràcters seria és el següent:

```
DataOutputStream sortida = new DataOutputStream(socket.getOutputStream());
sortida.write('a');
sortida.writeUTF("Aquest és un missatge per al servidor.");
```

Una altra de les classes que ens pot ser d'utilitat a l'hora d'escriure via *sockets* i que també és un envolvent de la superclasse `OutputStream` és la classe **`PrintWriter`**, encarregada d'imprimir les representacions formatades d'objectes cap a un flux de caràcters. Els mètodes més destacats són els següents:

- `PrintWriter(OutputStream out)`: constructor de la classe. Implícitament convertirà caràcters a bytes abans d'escriure-ho.

- `PrintWriter(Writer out)`: constructor de la classe. Tant en aquesta opció com en l'anterior, com a segon paràmetre li podem passar un booleà per configurar l'*autoflush*, és a dir, forçar l'escriptura de tot el que hi hagi al *buffer*. En cas contrari, no es fa.

- `print(boolean b)`, `print(char c)`, `print (int i)`, etc. Els caràcters es converteixen a bytes segons el sistema de codificació per defecte, i s'escriuen pel flux de sortida corresponent. Hi ha les mateixes variants estan per la crida `println()` que afegeix la línia actual escrivint un final de línia.

A part, com en el cas anterior, tenim la crida `write()` sobre enters, caràcters i cadenes de caràcters *(string)*, `flush()`, `close()`...

```
PrintWriter pw = new PrintWriter(socket.getOutputStream(),true);
pw.println("Aquest és el missatge que enviem al servidor.");
```

5.3.2. Llegir dades

De manera paral·lela, resumirem breument les principals classes i mètodes involucrats en la lectura de dades, tant en clients com en servidors, en comunicacions orientades a la connexió. Per llegir dades binàries a baix nivell (*array* de *bytes*), s'empra una instància de la superclasse `InputStream`. Les funcions principals són les següents:

- `InputStream()`: constructor de la classe.

- `int read()`: llegeix el següent byte el flux de dades (0...255). Si no hi ha dispobible cap byte perquè ens trobem al final del flux, retorna el valor -1. Aquest mètode és bloquejant, és a dir, el procés es quedarà en aquesta instrucció esperant a una nova dada.

- `int read(byte[] b)`: llegeix un nombre de bytes determinat per la llargada de la variable que es passa com a paràmetre. S'emmagatzemen a la posició *b[0]* fins la *b[len-1]*. Retorna el nombre de bytes que s'han llegit.

- `int available()`: retorna el nombre de bytes que es poden llegir al flux d'entrada.

- `close()`: tanca el flux d'entrada i allibera els recursos de sistema associats.

L'excepció més rellevant és:

- `IOException`: si ocorre qualsevol error d'entrada sortida.

Un exemple de codi seria:

```
InputStream entrada = socket.getInputStream();
byte[] dades = new byte[10];
int l = entrada.read(dades);
```

Si volem llegir dades a més alt nivell (caràcters o *strings*) podem utilitzar un envolvent de la superclasse `InputStream`, i convertir-lo a un objecte **`DataInputStream`**, de manera similar a com hem explicat en l'escriptura. Aquesta classe permet llegir tipus de dades primitius d'un flux d'entrada, heretada de la classe `DataInput`, que representa cadenes de caràcters codificades en un format Unicode que és una lleugera modificació del UTF-8. A més del mètodes de lectura comentats en la superclasse anterior, podem destacar:

- `DataInputStream(InputStream out)`: constructor de la classe.
- `byte readChar()`: llegeix 2 bytes del flux d'entrada i els converteix a un caràcter.
- `String readUTF()`: llegeix en una cadena de caràcters Unicode, codificada segons el format UTF-8 modificat.
- `int readInt()`: llegeix 4 bytes del flux d'entrada i els converteix a un enter.
- `readDouble`, `readFloat`, `readLong` ...

Una altra classe per a fer la lectura és **`InputStreamReader`**, interessant perquè converteix de manera transparent al usuari els bytes llegits en el flux d'entrada a caràcters, utilitzant la codificació que s'especifiqui o la que hi hagi configurada per defecte a la màquina. Els mètodes més rellevants són els següents:

- `InputStreamReader(InputStream out)`: constructor de la classe.

- `InputStreamReader(InputStream out, String charsetName)`: constructor de la classe, on li especifiquem la codificació que es farà servir en el procés de lectura de bytes des del flux d'entrada.

- `int read()`: lectura d'un sol caràcter.

- `int read(char[] cbuf, int offset, int len)`: lectura d'una cadena de caràcters de longitud *len*, emmagatzemats a partir de la posició *offset* del primer paràmetre.

- `close()`: tanca el flux d'entrada i allibera els recursos de sistema associats.

Un exemple per llegir un únic caràcter és el següent:

```
InputStream entrada = socket.getInputStream();
InputStreamReader lector = new InputStreamReader(entrada);
int c = lector.read();
System.out.println("Dades rebudes " + (char)c);
```

Si emprem la classe `InputStreamReader` i llegim byte a byte, el procés de conversió faria la nostra aplicació poc eficient. Per aquest motiu, és habitual emprar un nou embolcall com la classe **`BufferedReader`**, que utilitza *buffers* de lectura per a anar emmagatzemant el que hi ha disponible i anar consumint-ho segons es demani, mitjançant la tècnica FIFO. A part dels mètodes de lectura descrits en la classe anterior tenim els següents:

- `BufferedReader(Reader in)` : constructor de la classe. Com a segon paràmetre es pot indicar la mida del *buffer*.

- `String readLine()` : llegeix una línia de text, considerant com a final de línia \n, \r o \r\n.

Un exemple pot ser el següent:

```
InputStream entrada = socket.getInputStream();
BufferedReader lector = new BufferedReader(new InputStreamReader(entrada));
String linia = lector.readLine();
System.out.println("Dades rebudes " + linia);
```

5.4. Exemple d'un client i un servidor

A continuació, es mostra un exemple complet i funcional d'un client i un servidor orientats a la connexió. El funcionament general és el següent:

- El client envia al servidor tot el que l'usuari li introdueix per teclat, fins que rep la paraula *END*.

- El servidor llegeix tot el que li envia el primer client connectat i ho mostra per pantalla. Quan rep el missatge *END* d'aquest client, finalitza la execució.

És aconsellable que copieu el codi del servidor en un arxiu *servidortcp.java* i el codi del client en un arxiu *clienttcp.java*. Els podeu compilar i executar per provar la interacció entre client i servidor. Una execució tant en client com en servidor seria la següent:

```
escrivint
al
servidor
end
END
```

```
/** SERVIDOR ORIENTAT A LA CONNEXIÓ **/
/** Llibreries habituals quan treballem amb sockets **/
import java.net.*;
import java.io.*;
/** Es crea una nova classe Java anomenada Servidor TCP i una nova funció main que en governa
    el funcionament. En primer lloc es declaren dues variables, una de tipus socket Servidor
    TCP i l'altra com a missatge per a gestionar les dades que intercanviarem amb els clients
     potencials **/
public class servidortcp {
    public static void main(String argv[]) throws IOException {
        ServerSocket socket = null;
        String missatge;
/** Es crea un socket servidor en TCP anomenat socket pel port 6001 que espera connexions de
    clients potencials. Quan un client concret es connecta al servidor, s'accepta i es crean
    un nou socket anomenat socket_cli, mitjançant el qual s'estableix la comunicació
    bidireccional entre servidor i client **/
        socket = new ServerSocket(6001);
        Socket socket_cli = socket.accept();
/** Es declara i s'inicialitza una variable anomenada entrada per on es llegeixen les dades que
ens envia el client mitjançant el socket que s'hi ha establert. **/
DataInputStream entrada = new DataInputStream(socket_cli.getInputStream());


/** S'inicia l'intercanvi de missatges mitjançant un bucle que llegeix el que el client envia i
    ho mostra per pantalla fins que el client envia la paraula "END" **/
        do {
            missatge = entrada.readUTF();
            System.out.println(missatge);
        } while (!missatge.startsWith("END"));
/** Tanquem el socket que ens permetia comunicar-nos amb el client i el socket que estava a
    l'espera de noves connexions d'altres clients **/
        socket_cli.close();
        socket.close();
    }
}
```

```
/** CLIENT ORIENTAT A LA CONNEXIÓ **/
/** Llibreries habituals quan treballem amb sockets **/
import java.net.*;
import java.io.*;
/** Es crea una nova classe Java anomenada Client TCP i una nova funció main que en governa
    el funcionament. En primer lloc es declaren les variables, una de tipus socket que
    representarà un socket client TCP; l'altra anomenada adr per a gestionar l'adreça del
    servidor a la qual ens volem connectar; finalment la variable missatge s'utilitza per a
    gestionar les dades que intercanviem amb el servidor **/
public class clienttcp {
    public static void main(String argv[]) throws IOException {
        Socket socket = null;
        InetAddress adr;
        String missatge = "";
/** Es crea un socket client anomenat socket que es connectarà al servidor que li passem
    com a paràmetre del programa (argv[0]) i port 6001. **/
        adr = InetAddress.getByName(argv[0]);
        socket = new Socket(adr, 6001);
/** Es declara i s'inicialitza una variable anomenada entrada que llegirà el que l'usuari li
    escrigui per teclat i la variable sortida que serà l'encarregada d'escriure els missatges
  al servidor. **/
        BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
        DataOutputStream sortida = new DataOutputStream(socket.getOutputStream());
/** S'inicia l'intercanvi de missatges mitjançant un bucle que llegeix el que l'usuari introdueix
    per teclat i li envia al servidor, fins que escrivim la paraula "END" **/
        do {
            missatge = entrada.readLine();
            sortida.writeUTF(missatge);
        } while (!missatge.startsWith("END"));
/** Tanquem el socket client que ens permetia comunicar-nos amb el servidor **/
        socket.close();
    }
}
```

6. Altres operacions

A continuació, es llisten breument altres operacions auxiliars per a obtenir informació del *socket*. Algunes són exclussives dels servidors, altres poden ser emprades tant en client com en servidor.

Tant en la classe *Socket* com en la classe *ServerSocket*, destaquem les següents:

- `InetAddress getInetAddress()`: retorna d'adreça local del *socket*.
- `int getLocalPort()`: retorna el número de port pel qual el *socket* està escoltant.
- `SocketAddress getLocalSocketAddress()`: retorna l'adreça del *socket* local.
- `setReuseAddress(boolean on)`: activar o desactivar l'opció SO_REUSEADDR. Si no està activada, quan una connexió TCP es tanca, la connexió es troba en un estat de *timeout* durant cert temps (conegut com l'estat TIME_WAIT), cosa que fa que no pugui reutilitzar-se la mateixa adreça i port. De fet, per a aplicacions que utilitzen adreces o ports coneguts, probablement no serà possible aquesta reutilització.
- `setReceiveBufferSize(int size)`: especifica la mida del *buffer* de clients esperant a ser acceptats pel servidor, segons l'opció SO_RCVBUF.
- `int getReceiveBufferSize()`: obté el temps anterior.
- `boolean isClosed()`: retorna l'estat del *socket*, si està o no tancat.

Exclussives de la classe *Socket* (y, en conseqüència, no es poden fer servir en el *socket* d'escolta del servidor), destaquem les funcions següents:

- `int getPort()`: retorna el número de port remot amb el que ens estem comunicant.
- `SocketAddress getRemoteSocketAddress()`: retorna l'adreça del *socket* remot.

A continuació, presentem un exemple amb algunes d'aquestes funcions. Bàsicament, es crea un *socket* client que s'intenta conectar al port 80 (web) del servidor passat com a paràmetre, i mostra per pantalla totes les dades de la connexió:

```
import java.io.*;
import java.net.*;
public class informacioSocket {
    public static void main(String[] args) {
        try {
            Socket sc = new Socket(args[0], 80);
            System.out.println("Connectat al host " + args[0]
                + " amb adreça IP " + sc.getInetAddress().getHostAddress()
                + " i port " + sc.getPort()
                + ", des del meu socket amb adreça " + sc.getLocalAddress()
                + " i port " + sc.getLocalPort());
        }
        catch (UnknownHostException ex) {
            System.err.println("Error host desconegut " + args[0]);
        } catch (SocketException ex) {
            System.err.println("Error al connectar al host " + args[0]);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

El resultat esperat, si no es llança cap excepció durant la creació del *socket*, és:

```
Connectat al host uoc.edu amb adreça IP 213.73.40.242 i port 80,
des del meu socket amb adreça /192.168.1.137 i port 55102
```

Resum

Els *sockets* són una interfície de programari formada per un conjunt d'instruccions de codi, que els programes que es comuniquen per la xarxa han de seguir, perquè Internet pugui entregar les dades. A continuació, s'exposen resumidament les principals classes i mètodes per a programar una aplicació en xarxa en el llenguatge de programació Java.

En comunicacions no orientades a la connexió, es crea un únic *socket* en el servidor seqüèncial i un *socket* en cada un dels clients. Els dos són del mateix tipus, s'empra la classe `DatagramSocket` per a crear-los. La unitat d'intercanvi de dades és el datagrama i es representa per la classe `DatagramPacket`. Per a enviar i rebre dades pel *socket* client i servidor es fan servir els mètodes `send()` i `receive()`, que han d'indicar sempre les dades del *socket pair*.

En comunicacions orientades a la connexió, el servidor seqüencial crearà dos tipus de *sockets*. D'una banda, un *socket* d'escolta, amb nom conegut, representat per la classe *ServerSocket*, per on acceptarà clients, emprant el mètode `accept()`. D'altra banda, en el servidor també es crearà un socket per atendre a cada client que es connecti, sense nom conegut, representat per la classe *Socket*. En el client es crea el mateix tipus de *socket* de la classe *Socket*. Pel canal de comunicació bidireccional i unívoc que forma el *socket pair* entre servidor i cada client, i que es manté actiu mentre duri la comunicació, s'intercanvien fluxos de dades. Per a enviar i rebre dades, farem servir els mètodes `read()` i `write()` o similars de les superclasses `InputStream` i `OutputStream`. Per tal de facilitar la programació i el tractament de les dades, és habitual emprar classes envolvents o *wrappers* i els seus mètodes, com `DataInputStream`, `BufferedReader` o `DataOutputStreamPrintWriter`.

En qualsevol dels tipus de *sockets*, sempre és aconsellable tancar el *socket* utilitzant el mètode `close()`, quan ja no s'hagi de fer servir. D'aquesta manera, allibrerem recursos innecessaris del sistema i podem reutilitzar les connexions.

Bibliografia

Kurose, J., i Ross, K. (2000). *Computer Networking: A Top-Down Approach*. Boston: Pearson.

Webgrafía

<https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/howdosockets.htm>.

<https://docs.oracle.com/javase/tutorial/networking/overview/networking.html>.

<http://web.mit.edu/6.031/www/sp19/classes/23-sockets-networking/>.

<https://ioc.xtec.cat/materials/FP/Materials/2201_SMX/SMX_2201_M05/web/html/index.html>.

<https://www3.uji.es/~belfern/libroJava.pdf>.