els identifica anomenada MAC (*Media Access Control*). En una xarxa local la informació s'envia encapçalada per l'adreça MAC a tots els dispositius connectats. Els dispositius amb una adreça diferent descartaran la informació. En canvi el dispositiu destinatari la tractarà i la farà arribar a les capes superiors.

Les xarxes de gran abast són aquelles que aglutinen molts dispositius diferents. Per exemple les companyies de telecomunicacions disposen de xarxes de gran abast, però també els governs, les institucions i organitzacions internacionals disposen de grans xarxes interconnectades entre elles constituint el que coneixem com a Internet. Alguns dels protocols de xarxa usats aquí són PPP (*Pear to Pear Protocol*) que representen el punt d'accés a les xarxes de gran abast de dispositius particulars i xarxes d'àmbit local o ATM per aconseguir els enllaçaments intermedis.

1.4 Elements de programació d'aplicacions en xarxa

Els llenguatges d'alt nivell disposen de biblioteques especialitzades en el desenvolupament d'aplicacions distribuïdes. Malgrat que cada llenguatge contempla les seves particularitats, tots ells presenten força elements comuns que donen resposta als conceptes bàsics d'adreçament, enviament d'informació, connexió i canal de transmissió o tractament de recursos remots de la forma més transparent possible.

Implementació de capes inferiors

És important saber que Java només permet la implementació a la capa d'aplicació perquè aquesta característica descarta aquest llenguatge per implementar solucions alternatives de baix nivell. En general Java és un llenguatge destinat a implementar solucions d'alt nivell.

El llenguatge Java aprofita la riquesa que li dóna el paradigma orientat a objectes per definir una jerarquia de classes que embolcalla tota aquesta potencialitat a través de mètodes de molt alt nivell que faciliten el desenvolupament d'aplicacions distribuïdes robustes. De fet, l'API que contempla la biblioteca estàndard de Java només permet treballar a nivell de la capa d'aplicació usant la interfície d'accés exclusiu a la capa de transport. El JDK estàndard no contempla la possibilitat d'accedir a les capes de més baixes (xarxa o enllaç).

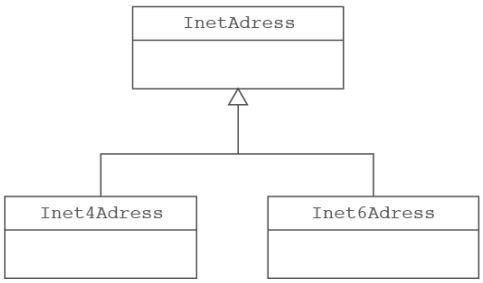
Es tracta d'una implementació realment eficient que dóna resposta a totes les necessitats originades des de la capa d'aplicació. Per tant no és necessari que ens plantegem si hem de fer servir altres alternatives, ni molt menys d'implementar-les nosaltres.

1.4.1 Adreçament

La classe `InetAddress` és una abstracció que ens permet gestionar de forma transparent adreces IP de qualsevol de les dues versions sense necessitat de tenir-ho en compte. Internament sempre es treballarà amb la classe corresponent a la versió adient (IPv4 o IPv6), que quedarà amagada al programador per la classe `InetAddress`. Això facilita molt la codificació ja que ens permet fer el mateix tractament amb independència de la versió IP usada.

Com a conseqüència d'aquesta abstracció, mai hauríem d'instanciar directament un objecte de la jerarquia invocant la sentència `new`, sinó que cal fer-ho a través d'un dels mètodes `static` que la classe `InetAddress` posa a la nostra disposició. Els més comuns són `getByName`, `getAllByName` `getByAddress` `getLoopbackAddress` i `getLocalHost`. Tots ells de cara al programador retornen una instància d' `InetAddress`, però internament si la IP utilitzada és de tipus IPv4 la instància serà de la classe `Inet4Address` però si és de tipus IPv6 serà de la classe `Inet6Address` (vegeu jerarquia a la figura 1.7).

FIGURA 1.7. Jerarquia de classes per gestionar l'adreçament a dispositius remots



El mètode `getByName` necessita un paràmetre de tipus `string`. El paràmetre podrà tractar-se d'una URL, d'un nom de xarxa identificador d'un dispositiu o d'una adreça IP en qualsevol dels formats acceptats pel protocol IP. En cas de tractar-se d'una cadena URL, la classe `InetAddress` farà una petició al servidor DNS per defecte, per aconseguir resoldre el nom subministrat i obtenir una adreça IP equivalent. Si la cadena passada conté un nom de xarxa local, la classe `InetAddress` usarà els protocols estàndards que permetin resoldre el nom i obtenir l'adreça IP associada. Les adreces IP obtingudes es convertiran a un vector de bytes de la mida corresponent a la versió de l'adreça (4 o 16 bytes). També, en cas que el paràmetre passat contingui directament una adreça IP, es farà la conversió a vector de bytes. El vector de bytes servirà per instanciar un objecte de tipus `Inet4Address` o `Inet6Address` d'acord amb la mida del vector. La instància creada es retornarà tot seguit com a resultat del mètode. En cas que el sistema usat per resoldre la URL o el nom de xarxa obtingui més d'una IP es crearà la instància d'una d'elles.

El mètode `getAllByName` rep també un paràmetre contenint una cadena amb les mateixes opcions que el mètode `getByName` i es comportarà de forma similar a l'hora de resoldre el nom passat, però si la resolució conté diverses adreces IP crearà una instància diferent per a cada una d'elles, les agruparà en un vector d'objectes `InetAddress` i les retornarà.

El mètode `getByAddress` espera rebre per paràmetre un vector amb 4 o 16 bytes. En cas que la mida del vector sigui de 4 bytes instanciarà un objecte de la classe `Inet4Address`. Seria de la classe `Inet6Address` si la mida fos de 16 bytes.

El mètode `getLoopbackAddress` obté una instància d' `InetAddress` fent servir l'adreça específica 127.0.0.1. De forma similar el mètode `getLocalHost` obté una instància de la connexió local principal de l'ordinador en el qual l'aplicació s'estigui executat.

En cas que el nom del paràmetre no es pugui resoldre o el format de l'adreça IP sigui incorrecta es llançarà una excepció del tipus `UnknownHostException`.

Les instàncies d' `InetAddress`, siguin de tipus `Inet4Address` o `Inet6Address` permeten totes obtenir informació del *host* associat a la IP. Entre d'altres el mètode `getAddress` ens retornarà un vector de bytes amb el valor de la IP associada. De forma semblant `getHostAddress` retorna també la IP però en format textual i el mètode `getHostName` obté el nom del dispositiu associat. Si durant la consulta el dispositiu no indiqués el nom, el mètode retornaria l'adreça IP en forma d'*string*.

En el següent exemple posem en pràctica l'ús d'alguns mètodes:

```
1  InetAddress[] addresses = new InetAddress[2];
2
3  addresses[0] = InetAddress.getLoopbackAddress();
4  addresses[1] = InetAddress.getByName("ioc.xtec.cat");
5  for(InetAddress adress: addresses){
6      if(adress.isLoopbackAddress()){
7          System.out.println(adress.getHostName() + " té una adreça loopback");
8      }else{
9          System.out.println(adress.getHostName() + " no té una adreça loopback")
               ;
10     }
11 }
```

També cal destacar els mètodes que ens informen sobre quin tipus d'adreça IP conté l'objecte InetAddress. A l'exemple podeu veure l'ús del mètode `isLoopbackAddress` que ens informa si la IP és del tipus 127.0.0.x. De forma semblant `isSiteLocalAddress` ens indica si es tracta d'una adreça de xarxa privada com per exemple ho pot ser 192.168.4.18. El mètode `isMulticastAddress` ens indica també si la IP l'objecte és de tipus multicast.

Tim Berners-Lee. Font: Knight Foundation

El projecte WWW

Tim Berners-Lee és l'inventor del projecte Word Wide Web, més conegut per WWW o simplement web. Originàriament el projecte va néixer amb la intenció de permetre compartir documents i informació entre investigadors. La idea del WEB gira entorn al concepte d'*hipertex*, un enllaç capaç d'identificar a un recurs específic en un entorn web. Actualment Berners-Lee és professor del MIT i director del World Wide Web Consortium (W3C).

1.4.2 Referències remotes i obtenció de recursos

Moltes vegades les aplicacions necessiten aconseguir recursos emmagatzemats en algun dispositiu de la xarxa. Sovint els recursos es troben emmagatzemats en forma de fitxer, però podrien també trobar-se emmagatzemats en una base de dades o obtenir-se com a resultat d'un procés.

Berners-Lee defineix un **recurs** com «allò que té un identificador». Ens diu que malgrat que alguns recursos són virtuals i es troben a la xarxa, cal també considerar recursos les persones, les organitzacions, o d'altres entitats del món real. També aclareix que el concepte recurs no és pas sinònim d'entitat sinó més aviat una representació puntual d'una entitat o amb els seves paraules, «un mapa conceptual d'una entitat en un instant de temps». De fet els recursos poden romandre constants, les entitats no.

Per identificar un recurs del web, podem fer servir una URL (Uniform Resource Locator). Una URL és una cadena de caràcters única per a cada recurs diferent, que segueix unes determinades regles sintàctiques i conté informació d'on es troba

el recurs, de com es pot localitzar. En tractar-se d'una cadena única per a cada recurs, diem que a més de localitzar el recurs també l'identifica. Per això les URL es consideren també identificadors de recursos.

En general la sintaxi d'una URL segueix el format següent:

```
esquema://acces/nom_especific
```

En el qual `esquema` és una paraula composta exclusivament de lletres corresponents a l'alfabet anglosaxó. També admet els caràcters `:`, `.`, `-` i `+`. Representa el nom del protocol d'accés o del context en el qual es troba el recurs: HTTP, FTP, TELNET, FILE, JDBC... en són exemples.

`acces` respon a algun dels següent formats:

- dispositiu
- dispositiu:port
- nom_usuari:contrasenya@dispositiu
- nom_usuari:contrasenya@dispositiu:port

L'`acces` indica en quin dispositiu es troba el recurs i com hi podem accedir. També se'l coneix com a autoritat (en anglès *Authority*) perquè representa l'autoritat que té drets sobre el recurs. A la posició `dispositiu` podem escriure una adreça IP o el nom del domini, i subdomini si fos necessari, que identifica el dispositiu. La identificació del dispositiu és obligatòria, però tota la resta és opcional. En cas de no especificar el port pel qual accedir, per defecte es considerarà el port reservat per defecte al protocol que fem servir. Si l'accés és lliure no caldrà especificar ni el nom d'usuari ni la contrasenya. Però si cal autorització es pot afegir, tal com mostra el format.

El `nom_específic` és una cadena que identifica el recurs dins del dispositiu. Pot estar compost de dues parts separades per un caràcter `?`. La primera és obligatòria. En cas que la segona part fos opcional s'ometrà el caràcter `?`. És a dir que seguira algun d'aquest formats:

- `cami`
- `cami?altres_indicacions_de_cerca`

El `cami` estarà format per dígits i cadenes de caràcters de l'alfabet anglosaxó separades pel caràcter `/`. Representen una ubicació jeràrquica semblant als arbres de directoris dels sistemes de fitxers, perquè originàriament, el `cami` de la URL coincidia amb el `cami` del sistema de fitxers del servidor on es trobava emmagatzemat el recurs. Actualment, malgrat que pot coincidir, no sempre és així. De fet, pot només tractar-se d'una identificació lògica sense correspondència amb cap ruta real ni tan sols amb cap fitxer (les dades podrien estar emmagatzemades en una base de dades, però identificades per mitjà d'un `cami` lògic. Per indicar el

camí, s'accepten també caràcters especials com ., - o _ a més dels dígits i l'alfabet anglosaxó. Tot i així, la necessitat d'internacionalitzar el sistema ha obligat a cercar una forma d'incorporar més caràcters. En cas que sigui necessari afegir caràcters no contemplats, podrà indicar-se el seu codi hexadecimal precedit d'un símbol %.

La part anomenada `altres_indicacions_de_cerca` (i que en anglès es coneix com a *query*) es troba constituïda per un nombre variable de parelles d'atribut-valor. Cada parella es trobarà separada de la següent per un símbol `&`. L'atribut apareixerà sempre en primer lloc (més a l'esquerra) i el valor en segon lloc immediatament després del símbol = que farà de separador. Veiem algun exemple:

Exemples de URLs

http://www.xtec.cat/documents/fp/dam/curriculum.html.

Es tracta d'una referència al recurs ubicat al camí *documents/fp/dam/curriculum.html* al qual podem accedir en el domini www.xtec.cat i usant el protocol HTTP. La referència usarà el port per defecte, és dir el port 80 que és el port per defecte del protocol HTTP.

ftp://admin:secret@ftp.debian.com/readme.txt.

Aquí es veu un URL que referencia el recurs *readme.txt*, usant el protocol FTP. El recurs es troba ubicat en el dispositiu que respon al domini `ftp.debian.com`. Per aconseguir el recurs cal enviar també el *login* i contrasenya que a l'exemple responen als valors `admin` i `secret` respectivament.

file:///home/user/recordatori.pdf

Aquí el recurs es troba en un disc local des del qual es fa la consulta seguint el camí *home/user/recordatori.pdf*.

http://www.servidor.com/cercar?tipus=compres&inici=2010-03-01&final=2010-03-31

Cas que cerca i obté a un recurs que representen les compres realitzades durant el mes de març registrades al servidor del domini www.servidor.com. El protocol d'accés és HTTP.

Classes per treballar amb referències a recursos

El Java Development Kit (JDK) de Java disposa de dues classes per poder treballar fàcilment amb recursos remots identificats amb una URL. Ens referim a els classes `URL` i `URLConnection`. Es tracta de classes de molt alt nivell que hauríem de situar dins la capa d'aplicació. Això permet al programador oblidar-se dels protocols més baixos i centrar la seva atenció en els protocols propis de l'aplicació.

La classe `URL` contempla dues atribucions. D'una banda representa la cadena que identifica el recurs i disposa de mètodes per tractar-la i extreure'n informació i de l'altre un punter al contingut del recurs.

Els principals mètodes que faciliten el tractament de la cadena localitzadora destriant la seva composició són:

- `String getProtocol()`: retorna una cadena amb el nom del protocol que la URL indica.
- `String getAuthority()`: obté l'autoritat extreta de la cadena URL.

- `String getPath()`: és la part que hem anomenat `cami` a la representació esquemàtica de la sintaxi de la URL.
- `String getQuery()`: invocant aquest mètode obtindrem la part anomenada `altres_indicacions_de_cerca`. En aquesta cadena s'exclou l'interrogant que la separa de `cami`.
- `String getFile()`: retorna la concatenació entre getPath() i getQuery(), mantenint entre ambdues parts el caràcter separador (un interrogant).
- `String getPort()`: en executar-lo retornarà el port especificat a la URL. Si no s'especifica port, retornarà -1.
- `String toString()`: obté tota la cadena de la URL. Aquest mètode és útil perquè la classe disposa de diversos constructors als quals se'ls pot passar la cadena sencera o per parts. Si fos així el constructor concatenarà les diverses parts passades per paràmetre i les formatarà adequadament amb els símbols adequats que les han de separar.

Els principals constructors són:

- `URL(String protocol, String acces, int port, String nomEspecificDelRecurs)`: constructor d'URL que crearà la cadena localitzadora, concatenant el protocol, l'accés, el port i el nom específic de recurs.
- `URL(String protocol, String acces, String nomEspecificDelRecurs)`: constructor d'URL que crearà la cadena de localització, concatenant el protocol, l'accés i el nom específic de recurs.
- `URL(URL context, String nomRelatiuDelRecurs)`: aquest constructor instancia la URL amb dos paràmetres i permet expressar un adreçament relatiu. Així la cadena del segon paràmetre actuarà com un localitzador relatiu al context passat en el primer paràmetre.
- `URL(String cadenaUrl)`: instanciarà un objecte URL a partir de la seva cadena passada per paràmetre.

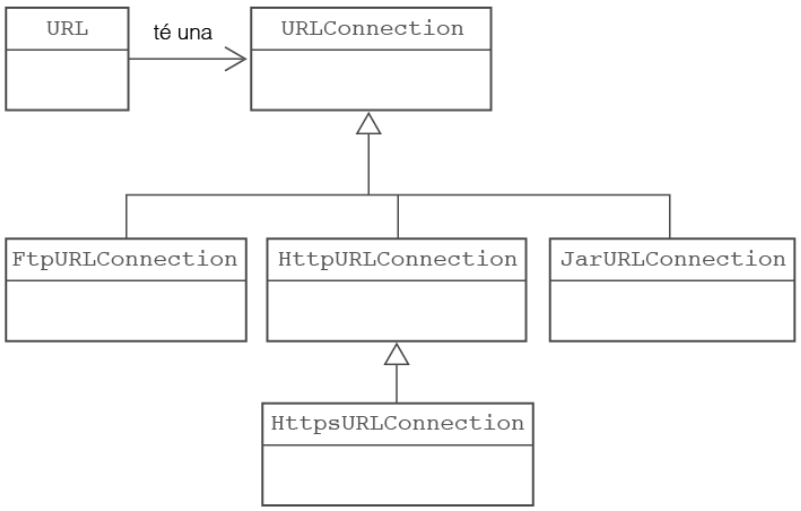
En referència a l'obtenció del contingut del recurs, podem invocar el mètode `openStream()` per aconseguir un flux d'entrada apuntant al contingut remot. És possible també obtenir directament una instància de `URLConnection` per disposar de més control sobre la connexió invocant el mètode `openConnection()`. La classe URL fa servir una instància com aquesta durant la invocació d' `openStream` per tal de poder retornar el flux de dades.

La classe `URLConnection` usarà el protocol TCP per fer la petició del recurs. Aquí el flux de dades cal obtenir-lo cridant `getInputStream()`. Amb aquesta classe tindrem a la nostra disposició alguns mètodes útils per obtenir informació del contingut. Cal però clarificar que es tracta d'informació extra (en forma de capçalera) aportada pel servidor al qual se li ha fet la petició del recurs. Podem

trobar-nos, doncs, que alguna vegada la informació no està del tot completa o fins i tot podria arribar a ser errònia. Entre d'altres, podem demanar la longitud del contingut (fent servir el mètode `getContentLength`), la codificació usada per emmagatzemar els caràcters o bé el tipus de dades que el recurs conté (imatge, text, XML, PDF, etc). La invocació necessària per obtenir el tipus de recurs és `getContentType` i per conèixer la codificació dels caràcters és `getContentEncoding`.

Tot i que la classe `URLConnection` pot suportar URLs de diversos protocols, el seu disseny està especialment pensat per donar suportar al protocol HTTP. És a dir, els seus mètodes suporten la funcionalitat descrita en el protocol HTTP, tot i que és capaç de treballar amb recursos remots apuntats per URLs que facin servir altres protocols. En realitat, `URLConnection` és una classe abstracta que delega la seva funcionalitat en les classes que l'estenen. Cada classe hereva d' `URLConnection` controlarà un protocol diferent (vegeu la figura 1.8). D'aquesta manera resulta força senzill afegir nous protocols o modificar els existents sense repercutir en la resta de classes.

FIGURA 1.8. Jerarquia de les classes que representen recursos remots



A nivell d'aplicació normalment treballarem sempre amb la classe `URLConnection` i no amb les seves derivades perquè així podem despreocupar-nos del protocol utilitzat i donar un tractament estàndard a totes les connexions. Quelcom semblant passa amb els objecte de tipus flux que seran retornats per aconseguir el contingut remot. La classe usada dependrà del tipus de contingut segons sigui imatge, PDF, text pla, o un document HTML, però pel programador totes elles caldrà tractar-les com una instància de la classe `InputStream`. Es tracta d'uniformitzar el tractament de tot el contingut tant com sigui possible.

Atenent que la classe `URLConnection` està orientada específicament al protocol HTTP, disposa d'un conjunt de mètodes que a més de recuperar el contingut, permeten obtenir informació extra definida en aquest protocol o bé enviar informació específica seguint les especificacions del protocol. Cal veure que es tracta de processos força genèrics comuns a diferents protocols i per tant és

possible reutilitzar la mateixa funcionalitat malgrat que estiguem fent servir altres protocols.

Recordeu que existeix una gran diversitat de formes de codificar els caràcters (ISO-8859-1, WINDOWS-1252, UTF-8...). Això és així perquè no existeix un únic model per representar qualsevol tipus de caràcter sinó que cada codificació és capaç de representar només un subconjunt més o menys ampli de caràcters. Actualment el sistema més universal és UTF.

Recordeu que la forma de representar una data és a partir d'un valor numèric que indica el nombre de mil·lisegons transcorreguts des del l'1 de gener de 1970.

D'acord amb l'especificació HTTP les dades viatgen embolcallades en una capçalera que conté metainformació del contingut. Generalment aquesta informació l'omple el servidor en el qual es troba ubicat el recurs. Es tracta d'informació opcional i per tant podem usar-la però cal tenir en compte que alguns recursos arribaran sense ella. Alguns d'aquests mètodes són:

- `String getContentEncoding()`. Obté la codificació de caràcters que el contingut del recurs fa servir.
- `int getContentLength()`. Aconsegueix la longitud del contingut que indica la capçalera.
- `String getContentType()`. Retorna el tipus de contingut d'acord amb l'estàndard HTTP. Aquesta especificació per exemple tipifica un document de text com a `text/plain`, un document HTML com a `text/html`, una imatge JPG com a `image/jpg`, etc.
  - `long getDate()`. Ofereix la data de creació/modificació del recurs con un valor numèric de tipus `long`

En cas que el tipus de contingut no es trobi informat a la capçalera podem fer servir el `guessContentTypeFromName` que intenta esbrinar el tipus a partir del nom del fitxer contingut a la URL. Aquest mètode pot servir per verificar o complementar la informació referent al tipus de dades del recurs.

Codi per comprovar si un recurs és una imatge de format GIF

Veieu a continuació un exemple que determina si un recurs en una imatge en format GIF comparant el valor retornat per els mètodes `getContentType` i `guessContentTypeFromName`. Fixeu-vos bé com se li passa el nom del fitxer fent servir el mètode de la classe URL pertinent.

```
1  boolean isGifFormat(URL url){
2      boolean ret = false;
3      try {
4          URLConnection con = url.openConnection();
5          String headerType = con.getContentType();
6          String guessType = con.guessContentTypeFromName(url.
               getFile());
7          ret = headerType.endsWith("gif") || guessType.endsWith("
               gif");
8      } catch (IOException ex) {
9          Logger.getLogger(URLUsage.class.getName()).log(Level.
               SEVERE, null, ex);
10     }
11     return ret;
12 }
```

`URLConnection` ens oferirà un flux de de dades per obtenir del contingut del recurs invocant el mètode `getInputStream()`. Es tracta del mateix flux de dades que obtindríem fent servir directament el mètode `openInputStream` de la classe URL perquè internament aquesta fa servir una instància d' `URLConnection` per accedir al contingut del seu recurs. És a dir, per aconseguir una instància d'un flux d'entrada podeu fer servir indiferentment qualsevol de les dues formes:

```
1 InputStream in = url.openStream();
```

o bé,

```
1 InputStream in = url.openConnection().getInputStream();
```

Vegeu l'annex "Manipulació dels fluxos" en la secció Annexos.

Cal tenir en compte a l'hora de treballar amb fluxos que només admeten de ser recorreguts en una única direcció i un cop han estat recorreguts no es poden tornar a recórrer. Normalment farem servir un *array* de caràcters o bé bytes per anar extraient de cop gran quantitat del contingut del flux fins extreure'l tot. Escollirem el tipus de dada de l'*array* segons el que contingui el recurs. Si es tracta d'un recurs de text podem convertir la instància d' `InputStream` en un objecte de tipus `InputStreamReader` per tal de disposar de mètodes més adients per tractar caràcters.

Obtenció de dades a través d'un flux de dades

Exemple de codi per extreure la informació arribada a través d'un flux de dades (`InputStream`).

```
1  private void printContent(URL url){
2      InputStream in;
3      char[] cbuf = new char[512];
4      int caractersLlegits;
5
6      if(!isText(url)){
7          return;
8      }
9
10     try {
11         in = url.openStream();
12         InputStreamReader inr = new InputStreamReader(in);
13         while((caractersLlegits=inr.read(cbuf))!=-1){
14             String str = String.copyValueOf(cbuf, 0,
                       caractersLlegits);
15             System.out.print(str);
16         }
17         System.out.println();
18     } catch (IOException ex) {
19         Logger.getLogger(URLUsage.class.getName()).log(Level.
               SEVERE, null, ex);
20     }
21 }
```

La classe `URLConnection` disposa també d'una connexió per enviar informació cap al recurs. De fet, HTTP especifica mecanismes per enviar dades des del client al servidor, per exemple extraient-les d'un formulari, o enviant un fitxer. Sovint les dades a enviar al servidor es concatenen juntament amb la mateixa cadena URL, constituint la part que hem anomenat `altres_indicacions_de_cerca` i que en anglès es coneix com a `query`. Malgrat tot HTTP també és capaç d'acceptar dades posteriors a la connexió a través d'una cadena URL. És el que es coneix com a mètode POST perquè primer s'estableix la connexió i immediatament després es comencen a enviar dades. El protocol HTTP estableix la seqüència fixa de passes que cal seguir per fer l'enviament amb èxit.

Imaginem que disposem d'una URL vàlida anomenada *url*. En primer lloc, caldrà obtenir la connexió fent,

```
URLConnection con = url.getConnection();
```

Després, cal indicar a la connexió que tenim intenció d'enviar dades invocant el mètode `setDoOutput`

```
con.setDoOutput(true);
```

i començar el procés d'enviament,

```
/*Suposarem que volem enviar caràcters i per tant usarem el tipus
  *OutputStreamWriter per facilitar el tractament*/
OutputStreamWriter out = new OutputStreamWriter(con.getOutputStream());
out.write("nomPropietat=" + valorPropietat);
```

En acabar cal tancar el flux de sortida per assegurar que s'envien totes les dades i s'alliberen els recursos,

```
out.close();
```

Com és natural, aquest sistema por fer-se servir també quan treballem amb altres protocols, perquè el diàleg específic propi del protocol es realitzarà internament dins de la classe específica (`HttpURLConnection`, `HttpsURLConnection`, `FtpUrlConnection`...) en el moment de fer la connexió i l'enviament de les dades.

1.4.3 Sòcols

API és l'acrònim d'Application Programming Interface.

Els sòcols, en anglès *sockets*, són una Interfície de Programació d'Aplicacions (API) que permet a dues aplicacions intercanviar informació malgrat que s'executin en dispositius diferents. Representen la porta d'entrada i sortida a la xarxa i constitueixen la base de qualsevol aplicació distribuïda.

> La documentació oficial de Java defineix els **sòcols** com el punt final de la comunicació bidireccional entre dos programes que s'executen a la xarxa.

Recordeu que l'arquitectura de les xarxes defineixen una pila de capes des de l'aplicació al medi físic per tal de fer factible la transmissió de dades entre programes. Els sòcols estarien situats a la capa de transmissió i representarien el punt d'accés de les aplicacions a les capes inferiors del sistema.

Per motius de seguretat i robustesa, Java no disposa de cap més utilitat que permeti treballar directament a les capes inferiors, per tant els sòcols es converteixen en la utilitat de programació de més baix nivell del llenguatge Java. Això significa que no és possible fer implementacions de protocols que es trobin per sota de la capa de transmissió.

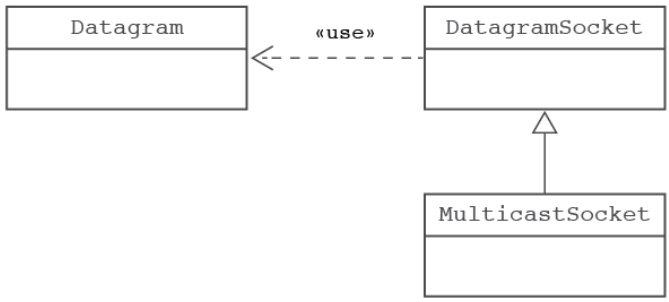
Els sòcols es troben associats a una IP i a un port de forma que sigui possible adreçar-hi informació a través de la xarxa fent servir algun dels protocols sobre IP disponibles (TCP o UDP).

> Segons el protocol utilitzat parlarem de sòcols **no orientats a connexió** quan utilitzin el protocol UDP i de sòcols **orientats a connexió** quan utilitzin el protocol TCP.

1.4.4 Implementació de sòcols no orientats a connexió

El llenguatge Java contempla 3 classes a l'hora de fer implementacions de comunicació no orientada a connexió. `DatagramSocket` i `DatagramPacket` constitueixen les classes bàsiques de la comunicació a través d'UDP. La tercera, deriva de `DatagramSocket` i dóna suport específicament a les comunicacions de tipus multicast (vegeu la figura 1.9).

Figura 1.9. Jerarquia de classes per implementar sòcols no orientats a connexió



DatagramSocket

Els sòcols creats usant `DatagramSocket` són capaços d'enviar i rebre els paquets especificats pel protocol UDP. En crear una instància podem especificar un port concret que el sòcol escoltarà quan sigui necessari atendre algun servei (estàndard o no) associat al port. Pels sòcols temporals, en canvi, no caldrà establir el valor del port. Durant la creació de la instància la classe cercarà el primer port lliure dins el rang disponible per atendre comunicacions temporals.

- `DatagramSocket()`. Genera una instància de `DatagramSocket` associada a un port temporal.
- `DatagramSocket(int port)`. Genera una instància de `DatagramSocket` associada al port que s'indica en el paràmetre.
- `DatagramSocket(int port, InetAddress)`. En dispositius que tinguin més d'una IP es podrà especificar a través del segon paràmetre la IP que es vincularà la instància generada.

No cal especificar sempre la IP en dispositius amb diverses adreces, la classe en seleccionarà una per defecte durant la creació de la instància.

Els objectes de la classe `DatagramPacket` representen paquets de dades d'acord a l'especificació definida pel protocol UDP. La classe és capaç d'afegir per defecte gran part dels camps obligatoris del paquet. Tot i així cal indicar els bytes de dades a enviar, la longitud d'aquestes i també l'adreça i el port destí.

- `DatagramPacket(byte[] bufferDeDades, int longitudDades, InetAddress adreca, int port).` Genera una instància de `DatagramPacket` que tindrà com a dades els bytes continguts al vector del primer paràmetre en la quantitat que indiqui el segon. Òbviament la longitud especificada no podrà superar mai la quantitat de bytes continguts al vector. El segon i tercer paràmetres permetran definir el destí de les dades.

Els objectes `DatagramSocket` disposen del mètode `send` per enviar un objecte `DatagramPacket` a l'adreça establerta en el propi paquet. A continuació veiem un exemple complet de creació d'instàncies i enviament d'un missatge de caràcters.

```
//bytes del missatge a enviar
byte[] missatge = "Salutacions".getBytes();
//adreça IP del destí
InetAddress adrecaDesti = InetAddress.getByName("localhost");
//port destí
int portDesti = 5555;
//creació del paquet a enviar
DatagramPacket packet = new DatagramPacket(missatge,
                                           missatge.length,
                                           adrecaDesti,
                                           portDesti);
//creació d'un sòcol temporal amb el qual realitzar l'enviament
DatagramSocket socket = new DatagramSocket();
//Enviament del missatge
socket.send(packet);
```

El procés de recepció de dades és molt similar, tot i que a l'hora de generar el paquet UDP cal crear-lo buit, però amb suficient capacitat per poder-hi copiar les dades rebudes.

```
//port a escoltar el servei que estem implementant
int portAEscoltar = 5555;
//vector de bytes en el qual rebre el missatge amb una capacitat de 1.024 bytes
byte[] missatge = new byte[1024];
//creació del paquet en el qual rebre les dades de 1.024 bytes com a màxim
DatagramPacket packet = new DatagramPacket(missatge,
                                           missatge.length);
//creació d'un sòcol que escolti el port passat per paràmetre
DatagramSocket socket = new DatagramSocket(portAEscoltar);
//recepció d'un paquet
socket.receive(packet);
```

Heu de tenir en compte, però, que rarament caldrà implementar un dispositiu que només hagi de fer enviaments o recepcions. Normalment caldrà definir petits diàlegs en els quals els dispositius hagin de rebre i enviar missatges successives vegades fins a completar l'objectiu de la comunicació. Els `DatagramSockets`, però, no són `full-duplex`, és a dir no poden processar al mateix temps un

enviament i una recepció, sinó que ho han de realitzar de forma seqüenciada, primer un procés i després l'altre.

Normalment un dels dispositius restarà escoltant a l'espera que d'altres iniciïn el diàleg. En general el dispositiu que escolta de manera indefinida fa el paper de servidor i els altres dispositius de client. Generalment una classe que implementi un servidor haurà de tenir algun mètode de configuració que permeti assignar el port que haurà d'atendre. També haurà de tenir un mètode que iniciï l'escolta del port, esperi la recepció de dades, les interpreti, n'obtingui la resposta, l'enviï al client que hagi fet la demanda i torni de nou a esperar una nova petició.

La implementació en Java del que acabem de descriure seria:

```java
public class DatagramSocketServer {
    DatagramSocket socket;

    public void init(int port) throws SocketException{
        socket = new DatagramSocket(port);
    }

    public void runServer() throws IOException{
        byte [] receivingData = new byte[1024];
        byte [] sendingData;
        InetAddress clientIP;
        int clientPort;

        //el servidor atén el port indefinidament
        while(true){
            //creació del paquet per rebre les dades
            DatagramPacket packet = new DatagramPacket(receivingData, 1024);
            //espera de les dades
            socket.receive(packet);
            //processament de les dades rebudes i obtenció de la resposta
            sendingData = processData(packet.getData(), packet.getLength());
            //obtenció de l'adreça del client
            clientIP = packet.getAddress();
            //obtenció del port del client
            clientPort = packet.getPort();
            //creació del paquet per enviar la resposta
            packet = new DatagramPacket(sendingData, sendingData.length,
                                        clientIP, clientPort);
            //enviament de la resposta
            socket.send(packet);
        }
    }

    private byte[] processData(byte[] data, int length) {
        //procés diferent per cada aplicació
        ...
    }
}
```

Fixeu-vos que es tracta d'una classe prou genèrica per adaptar-se a la majoria d'aplicacions, tan sols caldria adaptar el mètode `processData` als requeriments específics de cada implementació.

Fixeu-vos també, que la classe `DatagramPacket` disposa de mètodes per obtenir la informació continguda en el paquet. Això, a banda d'aconseguir les dades rebudes, permet també saber a quina adreça i port caldrà enviar la resposta generada.

De forma semblant al servidor, es pot definir un client de manera genèrica tenint en compte que és el client el que iniciarà la comunicació i per tant cal un procés

específic que aquí anomenarem `getFirstRequest` per tal d'aconseguir les dades inicials que caldrà enviar al servidor. A més, cada cop que el servidor ens enviï la resposta al nostre requeriment, caldrà fer-la arribar als interessats a través del mètode `getDataToRequest` que rebrà o bé un nou requeriment pel servidor o bé el senyal per finalitzar la connexió i abandonar l'execució de l'aplicació client.

Podeu imaginar que segons de quina aplicació client, `getDataToRequest` haurà de fer diferents coses amb les dades rebudes i obtenir de diferent manera les noves dades a enviar. Per exemple algunes aplicacions requeriran només mostrar les dades per pantalla i esperar la resposta de l'usuari, però d'altres potser requeriran també emmagatzemar les respostes en una fitxer o base de dades, obtenir la resposta mitjançant un procés automatitzat, etc. Sigui com sigui, el procés caldrà realitzar-lo sempre a través del mètode `getDataToRequest` per tal d'aconseguir un client prou adaptable.

El client també ha de ser capaç de reconèixer el senyal de finalització indicat per l'usuari, per exemple, en una de les seves respostes o com a resultat d'un procés que avalua que ja no cal continuar més la comunicació amb el servidor. En el nostre exemple aconseguim un procés neutre incloent el mètode `mustContinue` al qual se li passarà les dades retornades per `getDataToRequest` i ens retornarà si cal o no continuar la comunicació. Usarem aquest valor *booleà* per mantenir-nos dins del bucle o bé per abandonar-lo.

Abans d'iniciar la comunicació caldrà haver configurat el client indicant el nom del servidor i el port on es trobarà escoltant. Per això usarem el mètode `init`, malgrat que també es podria implementar en el propi constructor.

```
public class DatagramSocketClient {
    InetAddress serverIP;
    int serverPort;
    DatagramSocket socket;

    public void init(String host, int port) throws SocketException,
                                                   UnknownHostException{
        serverIP = InetAddress.getByName(host);
        serverPort = port;
        socket = new DatagramSocket();
    }

    public void runClient() throws IOException{
        byte [] receivedData = new byte[1024];
        byte [] sendingData;

        //a l'inici
        sendingData = getFirstRequest();
        //el servidor atén el port indefinidament
        while(mustContinue(sendingData)){
            DatagramPacket packet = new DatagramPacket(sendingData,
                                                       sendingData.length,
                                                       serverIP,
                                                       serverPort);
            //enviament de la resposta
            socket.send(packet);

            //creació del paquet per rebre les dades
            packet = new DatagramPacket(receivedData, 1024);
            //espera de les dades
            socket.receive(packet);
            //processament de les dades rebudes i obtenció de la resposta
```

```
            sendingData = getDataToRequest(packet.getData(), packet.getLength()
                );
        }
    }

    private byte[] getDataToRequest(byte[] data, int length) {
        //procés diferent per cada aplicació
        ...
    }

    private byte[] getFirstRequest() {
        //procés diferent per cada aplicació
        ...
    }

    private boolean mustContinue(byte[] sendingData) {
        //procés diferent per cada aplicació
        ...
    }
}
```

En aquesta implementació s'ha optat per rellançar tots els llançaments d'excepcions per tal de deixar el tractament d'errors pel darrer nivell d'implementació quan es pugui decidir què s'ha de fer amb el errors. Tot i així, hi ha una excepció que caldria controlar aquí. En cas que el servidor tingui problemes i no sigui capaç de contestar el requeriment d'un client, aquest podria romandre esperant la resposta indefinidament. És possible evitar aquesta situació fent servir el mètode `setSoTimeout` que limitarà el temps d'escolta només a la quantitat de mil·lisegons que se li passi com a paràmetre. Si el `DatagramSocket` sobrepassa el temps d'espera es llançarà una excepció de tipus `SocketTimeoutException`. Ara bé, no ens interessa que el llançament provoqui abandonar el bucle i per tant la finalització de l'execució, sinó que és preferible fer una notificació sense sortir del bucle `while`. El codi de l'interior del bucle podria quedar com segueix:

```
DatagramPacket packet = new DatagramPacket(sendingData,
                                           sendingData.length,
                                           serverIP,
                                           serverPort);
//enviament de la resposta
socket.send(packet);

//creació del paquet per rebre les dades
packet = new DatagramPacket(receivedData, 1024);
//espera de les dades, màxim 5 segons
socket.setSoTimeout(5000);
try{
    //espera de les dades
    socket.receive(packet);
    //processament de les dades rebudes i obtenció de la resposta
    sendingData = getDataToRequest(packet.getData(), packet.getLength());
}catch(SocketTimeoutException e){
    sendingData = timeoutExceeded(packet);
}
```

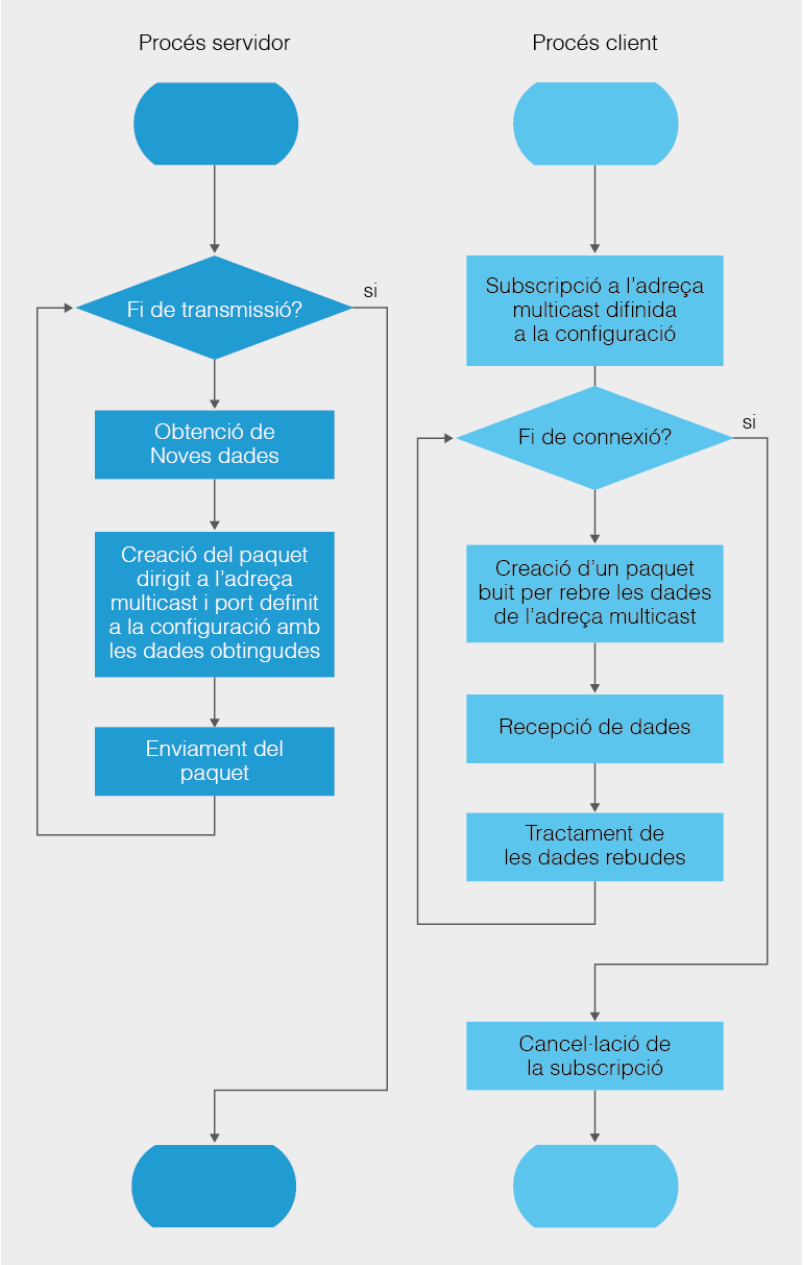
Per últim ambdues classes haurien de tenir també un mètode que permeti fer el tancament de l'objecte `DatagramSocket` abans d'acabar amb l'execució del servidor o del client. Per exemple:

```java
public void close(){
    if(socket!=null && !socket.isClosed()){
        socket.close();
    }
}
```

MulticastSocket

La jerarquia de sòcols UDP de Java contempla també la classe `MulticastSocket`. Amb aquesta classe és possible crear aplicacions que usin adreces multicast. Recordeu que les adreces multicast permeten als dispositius subscriure's a la difusió de tot el que arribi per l'adreça a la qual s'ha subscrit. Qualsevol objecte `MulticastSocket` podrà fer enviaments a tots els dispositius subscrits i rebre tot allò que vagi dirigit a l'adreça associada. Malgrat tot, generalment les aplicacions `multicast` fan els enviaments des d'un servidor i els clients subscrits es limiten a escoltar i processar les dades, de forma semblant al que es veu a la figura 1.10.

FIGURA 1.10. Diagrama de flux del procés d'un servidor multicast



L'esquema simula un servidor que només envia dades d'una manera contínua

L'ús de la classe `MulticastSocket` és molt semblant a la classe `DatagramSocket`, pel que fa a la recepció i l'enviament de dades perquè ambdues instancien els paquets com a objectes `DatagramPaquet` i fan servir el protocol UDP per a la transmissió. Tot i així, cal assenyalar que la classe `MulticastSocket` disposa d'un mètode per realitzar la subscripció (`joinGroup`) i un altre per cancel·lar-la (`leaveGroup`). A més destacarem que els sòcols Multicast han de treballar tots en el mateix port, ja siguin clients o servidors. Anem a veure com podem plasmar l'esquema anterior amb Java. Començarem codificant un servidor genèric. El mètode `init` permetrà configurar les instàncies per mitjà dels paràmetres, l'adreça multicast representada com un *string* i el port comú en el qual enviar totes les dades.

```java
public class MulticastSocketServer {
    MulticastSocket socket;
    InetAddress multicastIP;
    int port;
    boolean continueRunning = true;

    public void init(String strIp, int portValue) throws SocketException,
                                                          IOException{
        socket = new MulticastSocket(portValue);
        multicastIP = InetAddress.getByName(strIp);
        port = portValue;
    }

    public void runServer() throws IOException{
        DatagramPacket packet;
        byte [] sendingData;

        //el servidor fa enviaments continus mentre calgui
        while(continueRunning){
            //obtenció de les dades a enviar. Suposarem l'existència d'un
            //mètode específic per a les diferents aplicacions
            sendingData = getNextData();
            //creació del paquet per enviar les dades obtingudes
            packet = new DatagramPacket(sendingData, sendingData.length,
                                        multicastIP, port);
            //enviament de les dades
            socket.send(packet);

            //mirem si cal acabar la transmissió
            continueRunning = !transmissionFinished();
        }
    }

    public void close(){
        if(socket!=null && !socket.isClosed()){
            socket.close();
        }
    }
}
```

Destacarem que la funcionalitat del mètode `getData` dependrà de cada aplicació. També veiem que es contempla el mètode `close` per a preveure el tancament del sòcol abans de finalitzar l'execució del servidor.

El client, abans d'iniciar el bucle per rebre totes les dades, haurà de subscriure's a l'adreça multicast i en finalitzar el bucle caldrà cancel·lar la subscripció per reduir despeses inútils de recursos i tràfic a la xarxa innecessari.

```java
public class MulticastSocketClient {
    MulticastSocket socket;
    InetAddress multicastIP;
    int port;

    public void init(String strIp, int portValue) throws SocketException,
                                                          IOException{
        multicastIP = InetAddress.getByName(strIp);
        port = portValue;
        socket = new MulticastSocket(port);
    }

    public void runClient() throws IOException{
        DatagramPacket packet;
        byte [] receivedData = new byte[1024];
        boolean continueRunning = true;

        //activem la subscripció
```

```
19          socket.joinGroup(multicastIP);
20
21          //el client atén el port fins que decideix finalitzar
22          while(continueRunning){
23              //creació del paquet per rebre les dades
24              packet = new DatagramPacket(receivedData, 1024);
25              //espera de les dades, màxim 5 segons
26              socket.setSoTimeout(5000);
27              try{
28                  //espera de les dades
29                  socket.receive(packet);
30                  //processament de les dades rebudes i obtenció de la resposta
31                  continueRunning = getData(packet.getData(), packet.getLength())
                        ;
32              }catch(SocketTimeoutException e){
33                  //s'ha excedit el temps d'espera i cal saber què s'ha de fer
34                  continueRunning = timeoutExceeded();
35              }
36          }
37
38          //es cancel·la la subscripció
39          socket.leaveGroup(multicastIP);
40      }
41
42
43      public void close(){
44          if(socket!=null && !socket.isClosed()){
45              socket.close();
46          }
47      }
48
49  }
```

1.4.5 Implementació de sòcols orientats a connexió

Els sòcols orientats a connexió fan servir el protocol TCP. Recordeu que en aquest cas les dades es passen a l'aplicació en el mateix ordre en què han sortit i s'assegura que no es perd informació durant la transmissió. El protocol TCP defineix que abans de començar la transmissió de dades cal fer una petició de connexió que l'altre part haurà d'acceptar. Un cop acceptada la connexió, en ambdós costats es reservarà un port de xarxa exclusivament per a la transmissió de dades en qualsevol dels dos sentits. Recordeu que TCP és un protocol que defineix una comunicació *full-duplex* exclusiva entre dos dispositius.

Java té una manera pròpia d'implementar aquestes característiques. Dels dos dispositius a posar en contacte, un d'ells farà el paper de servidor esperant la demanda d'algun dispositiu, que jugarà el paper de client. El procés del servidor usarà una instància de la classe `ServerSocket` per restar escoltant fins que un client faci una petició, moment en què acceptarà la connexió i generarà una instància de tipus sòcol per mantenir la comunicació *full-duplex*, des d'un port temporal, durant la transmissió. El procés dels clients en canvi, usarà directament un objecte de la classe sòcol per fer la petició i també per mantenir la connexió i la transmissió de dades.

Per a la transmissió de dades es fa servir el concepte de flux de dades. Es considera que les dades flueixen de forma contínua i seqüencial des de l'origen fins al destí.

La característica de *seqüencial* és essencial aquí, ja que garanteix que l'ordre d'arribada a la destinació és el mateix que l'ordre de sortida des de l'origen.

El concepte de flux (*stream* en anglès) és una abstracció relacionada amb qualsevol procés de transmissió d'informació entre un origen i un destí.

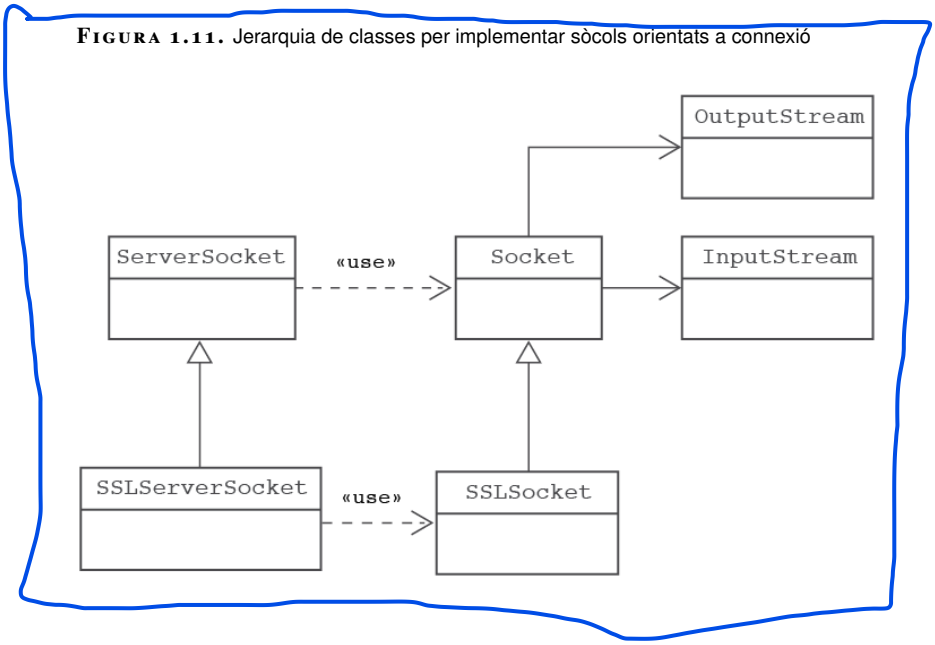
El flux de dades segueix un únic sentit (de l'origen al destí), per tant les aplicacions que necessitin transmetre i rebre hauran de fer servir dos fluxos, un en cada sentit.

El flux que rep dades des d'un origen extern s'anomena **flux d'entrada** (*input stream* en anglès) i el que envia informació a un destí exterior s'anomena **flux de sortida** (*output stream* en anglès).

Els objectes de tipus sòcol contenen dos fluxos de dades a través d'un d'ells es rebran les dades (flux d'entrada) i a través de l'altre s'enviaran (flux de sortida). En aplicacions client servidor, el flux de sortida del servidor equival al flux d'entrada del client i el flux d'entrada del servidor coincidirà amb el de sortida del client.

A la figura 1.11 podeu veure el diagrama de les classes que entren en joc en una transmissió TCP de Java. Les classe `SSLServerSocket` i `SSLSocket` ofereixen la mateixa funcionalitat que `ServerSocket` i `Socket` respectivament però estan especialitzades en transmetre de forma segura usant un protocol de seguretat anomenat SSL.

FIGURA 1.11. Jerarquia de classes per implementar sòcols orientats a connexió



La implementació que el llenguatge Java fa del protocol TCP passa per disposar en un dels dispositius a comunicar, el que faci el paper de servidor, d'una instància de `ServerSocket` que tindrà per objectiu únicament esperar que algun client decideixi fer una petició de comunicació, utilitzant els mínims recursos possibles mentre duri l'espera. El port a atendre, s'indicarà en el moment de creació de la instància de ServerSocket com a paràmetre del constructor.

Creació d'un ServerSocket que atendrà el port 7777 a l'espera de clients que demanin de comunicar-se

```
1         ServerSocket serverSocket = new ServerSocket(7777);
```

El client necessitarà una instància de Socket, la qual disposarà des del primer moment dels recursos propis del protocol TCP, ja que les instàncies de Socket solen crear-se indicant l'adreça i el port amb el que caldrà comunicar-se i des de la creació que es realitza la petició inicial per poder generar els recursos pertinents com els fluxos de dades que simularan els dos canals de comunicació que el protocol TCP demana.

Creació d'un sòcol en un client que vulgui comunicar-se amb un servidor que es trobi atenent el port 7777 a l'adreça 192.168.4.1

```
1         Socket socket = new Socket(new InetAddress.getByName("
              192.168.4.1"), 7777);
```

Durant la creació del sòcol del client s'enviarà al servidor la petició, el qual en rebre-la l'acceptarà creant un sòcol específic per comunicar-se amb el client acceptat a la banda del servidor. Recordeu que la comunicació TCP és exclusiva per dos dispositius, per això el servidor ha d'anar creant un sòcol específic per cada client que faci la petició.

La creació del sòcol a la banda del servidor es troba automatitzat i s'obté com a retorn del mètode de `ServerSocket` que inicia l'escolta del port fins l'arribada de la petició. Ens referim al mètode `accept`.

Mètode accept

```
1         Socket socket = serverSocket.accept();
```

El mètode accept resta a l'espera d'una petició i en el moment de produir-se crea una instància específica de sòcol per suportar la comunicació amb el client acceptat.

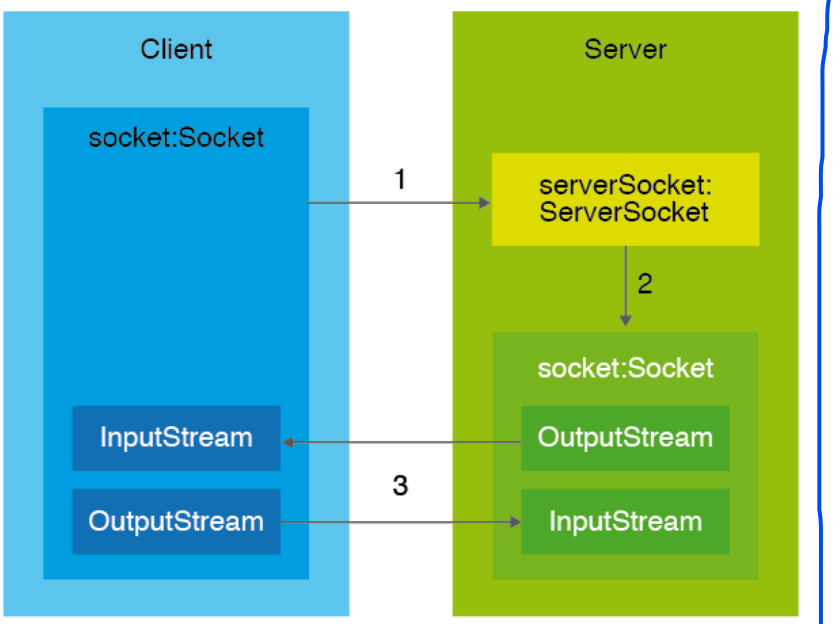
A partir d'aquí ambdós sòcols, el de la part client i el de la part servidor, disposaran d'un canal d'entrada i un de sortida adequadament connectats. La implementació dels canals es realitzarà a través d'instàncies internes de fluxos d'entrada i sortida. Per obtenir els fluxos d'un sòcol caldrà invocar els mètodes `getInputStream` i `getOutputStream` per obtenir respectivament els fluxos d'entrada i sortida.

Usarem els mètodes propis del fluxos per enviar o rebre dades a través d'ells. L'enviament a través del flux de sortida d'un dels sòcols implicarà la recepció de les mateixes dades enviades a través del flux d'entrada de l'altre sòcol. Podeu veure l'esquema del procés descrit a la figura 1.12 on:

- 1: El sòcol de la part client demana connectar-se a l'adreça en la qual es trobi el servidor i al port que escolti una instància de `ServerSocket` del servidor.
- 2: El `ServerSocket` accepta la connexió i crea un sòcol a la part del servidor amb dos canals, un d'entrada i un de sortida.
- 3: l' `InputStream` (flux d'entrada) del sòcol de la part client es troba connectat a l' `OutputStream` (flux de sortida) del sòcol de la part servidor

i el seu flux d'entrada connecta amb el flux de sortida del sòcol del client. D'aquesta manera és possible la comunicació en ambdues direccions amb la qualitat exigida pel protocol TCP.

FIGURA 1.12. Esquema del procediment seguit per connectar amb una servidor TCP de Java



A continuació veurem un exemple senzill de la implementació d'un diàleg entre un servidor i un client en el qual alternen consecutivament petició i resposta. Hem pres la consideració que tant les peticions com les respostes s'inclouen en una única línia de caràcters. D'aquesta manera cada part pot saber quan acaba l'enviament remot per tal de processar les dades i generar una resposta o una nova petició.

A la classe `TcpSocketServer` que representarà una aplicació de tipus servidor destacarem el mètode `listen` en el qual implementarem un bucle continuu en el qual una instància de ServerSocket anirà escoltant a cada iteració el port a atendre esperant clients que es connectin i acceptant les seves peticions a instanciar un sòcol. La comunicació entre ambdós objectes de tipus sòcol (el del client i l'acabat de generar) la implementarem en un mètode a part que anomenarem `proccesClientRequest`.

```
1  public class TcpSocketServer {
2      static final int PORT=9090;
3      private boolean end=false;
4
5      public void listen(){
6          ServerSocket serverSocket=null;
7          Socket clientSocket = null;
8          try {
9              serverSocket = new ServerSocket(PORT);
10
11             while(!end){
12                 clientSocket = serverSocket.accept();
13                 //processem la petició del client
14                 proccesClientRequest(clientSocket);
```

```
                //tanquem el sòcol temporal per atendre el client
                closeClient(clientSocket);
            }
            //tanquem el sòcol principal
            if(serverSocket!=null && !serverSocket.isClosed()){
                serverSocket.close();
            }

        } catch (IOException ex) {
            Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
                 null, ex);
        }
    }

    ...
}
```

El mètode per gestionar les peticions del client (`proccesClientRequest`) constarà d'un bucle continu que iterarem fins que el client ens indiqui que no té més peticions a realitzar. Per facilitar la dinàmica de pregunta resposta, convindrem que tant la petició com la resposta s'inclouran en un sola línia de caràcters. D'aquesta manera és fàcil detectar el final del missatge en qualsevol dels dos costats.

```
...
public void proccesClientRequest(Socket clientSocket){
    boolean farewellMessage=false;
    String clientMessage="";
    BufferedReader in=null;
    PrintStream out=null;
    try {
        in = new BufferedReader(new InputStreamReader(clientSocket.
            getInputStream()));
        out= new PrintStream(clientSocket.getOutputStream());
        do{
            //processem el missatge del client i generem la resposta. Si
            //clientMessage és buida generarem el missatge de benvinguda
            String dataToSend = processData(clientMessage);
            out.println(dataToSend);
            out.flush();
            clientMessage=in.readLine();
            farewellMessage = isFarewellMessage(clientMessage);
        }while((clientMessage)!=null && !farewellMessage);
    } catch (IOException ex) {
        Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}
...
```

Per més detalls, vegeu l'annex "Manipulació de fluxos" de l'apartat Annexos.

En tractar-se d'un diàleg basat en cadenes de caràcters delimitades per un final de línia, podem fer servir els fluxos específics orientats a caràcters que disposa el llenguatge. Recordeu que Java permet convertir qualsevol flux en un flux específic orientat a caràcters durant el procés de construcció.

El mètode `isFarewellMessage` indicarà si el missatge del client es correspon amb una petició de finalització de la comunicació. Si no s'hi correspon, caldrà processar la petició i obtenir la resposta mitjançant el mètode `processData`. La resposta obtinguda s'enviarà al client a través d'un *stream* de tipus `PrintStream` que afegirà al final de l'enviament un símbol de final de línia.

Forçarem l'enviament de les dades invocant el mètode `flush` i seguidament esperarem un nova petició del client, començant de nou l'anàlisi i si cal processant una nova iteració del bucle.

Per tal de gestionar de forma eficient els recursos del servidor, és important tancar les instàncies de sòcol dels clients que hagin demanat la finalització de la comunicació. Implementarem el tancament en un mètode específic anomenat `closeClient` que invocarem des del mètode `listen` tot just en acabar de retornar de la invocació del mètode `processClientRequest`.

```
...
private void closeClient(Socket clientSocket){
    //si falla el tancament no podem fer gaire cosa, només enregistrar
    //el problema
    try {
        //tancament de tots els recursos
        if(clientSocket!=null && !clientSocket.isClosed()){
            if(!clientSocket.isInputShutdown()){
                clientSocket.shutdownInput();
            }
            if(!clientSocket.isOutputShutdown()){
                clientSocket.shutdownOutput();
            }
            clientSocket.close();
        }
    } catch (IOException ex) {
        //enregistrem l'error amb un objecte Logger
        Logger.getLogger(TcpSocketServer.class.getName()).log(Level.SEVERE,
            null, ex);
    }
}
...
```

És important seguir la seqüència d'instruccions del mètode `closeClient` si volem assegurar una tancament correcte. Els mètodes `shutDownInput` i `ShutDownOutput` engeguen ambdues peticions de tancament necessàries en el protocol TCP. La invocació del mètode `shutDownInput` enviarà la petició de tancament al sòcol remot i esperarà la confirmació de la mateixa. El sòcol remot en rebre una demanda de tancament del seu canal de sortida forçarà un darrer enviament de les dades contingudes al *buffer* abans d'enviar la confirmació. Quan el servidor local rebi la confirmació alliberarà el bloqueig de lectura del seu `InputStream`. De forma semblant, la invocació de `ShutDownOutput`, provocarà el forçament de l'enviament de les darreres dades contingudes al *buffer* de sortida local i enviarà la petició de tancament del canal d'entrada del sòcol remot. El sòcol remot abans de confirmar el tancament alliberarà el bloqueig de lectura dels seu `InputStream`.

Un cop fetes les peticions de tancament i rebudes les confirmacions podrà processar-se el tancament local dels fluxos i seguidament el tancament del sòcol. Cal tenir en compte que els tancament dels objectes sòcol no impliquen el tancament del servidor, sinó que aquest restarà disponible per acceptar noves peticions ja que el "serverSocket" restarà obert en el bucle inicial del mètode "listen". Quan decidim fer el tancament de tot el servidor caldrà tancar també la instància del "serverSocket". A la part client, a la classe `TcpSocketClient` implementarem la connexió en el mètode `connect`. Aquí iniciarem un sòcol per tal que demani una petició de connexió TCP a l'adreça i port passada per paràmetre

i iniciarem un bucle continu que permeti anar fent peticions i rebent les respostes aplicant la mateixa consideració que ambdues aniran delimitades per un final de línia.

```
public class TcpSocketClient{

    public void connect(String address, int port) {
        String serverData;
        String request;
        boolean continueConnected=true;
        Socket socket;
        BufferedReader in;
        PrintStream out;
        try {
            socket = new Socket(InetAddress.getByName(address), port);
            in = new BufferedReader(new InputStreamReader(socket.getInputStream
                ()));
            out = new PrintStream(socket.getOutputStream());
            //el client atén el port fins que decideix finalitzar
            while(continueConnected){
                serverData = in.readLine();
                //processament de les dades rebudes i obtenció d'una nova
                    petició
                request = getRequest(serverData);
                //enviament de la petició
                out.println(request);//assegurem que acaba amb un final de lí
                    nia
                out.flush(); //assegurem que s'envia
                //comprovem si la petició és un petició de finalització i en
                    cas
                //que ho sigui ens preparem per sortir del bucle
                continueConnected = mustFinish(request);
            }

            close(socket);
        } catch (UnknownHostException ex) {
            reportError("Error de connexió. No existeix el host", ex);
        } catch (IOException ex) {
            reportError("Error de connexió indefinit", ex);
        }
    }
    ...
}
```

Per tal que funcioni correctament aquest algoritme serà necessari disposar d'un mètode especific anomenat `getRequest` que processi la resposta rebuda i aconsegueixi una nova petició que acabarà enviant-se al servidor. L'algoritme fa servir també el mètode `mustFinish` per detectar si la petició enviada implica una finalització de la connexió per tal de forçar la sortida del bucle.

Fora del bucle caldrà fer el tancament del sòcol de forma semblant de com s'ha realitzat en el servidor, assegurant l'ordre adequat de les peticions. És important comprovar si ja s'ha realitzat la petició del tancament de la connexió TCP usant els mètodes `isShutDownInput` o `isShutDownOutput` perquè no sabem quina de les parts (el servidor o el client) farà primer la petició.

```
...
    private void close(Socket socket){
        //si falla el tancament no podem fer gaire cosa, només enregistrar
        //el problema
        try {
            //tancament de tots els recursos
            if(socket!=null && !socket.isClosed()){
                if(!socket.isInputShutdown()){
```

```
                    socket.shutdownInput();
                }
                if(!socket.isOutputShutdown()){
                    socket.shutdownOutput();
                }
                socket.close();
            }
        } catch (IOException ex) {
            //enregistrem l'error amb un objecte Logger
            Logger.getLogger(TcpSocketClient.class.getName()).log(Level.SEVERE,
                 null, ex);
        }
    }
    ...
```