# INFORME PRÀCTICA 1
# XARXES II
# MÀSTER EN ENGINYERIA DE P. L.

Ignasi Barri Vilardell
Alberto Montañola Lacort
Josep Rius Torrentó

5 d'abril de 2007

# Índex



# Índex de figures



# Índex de taules

# 1 Introducció

En aquest informe s'intenta mostrar una guia de les decisions preses pels tres integrants del grup alhora de resoldre la pràctica 1 de l'assignatura de *Xarxes II* que està continguda dins del pla d'estudis del *Màster en Enginyeria de Programari Lliure* que s'imparteix en l'*Escola Politècnica Superior* dins de la *Universitat de Lleida*.

## 1.1 Pretencions inicials de la pràctica

La pràctica que s'ens proposa, tracta la problemàtica d'implementar un *xat*[1] usant una estructura de *client-servidor*, això sí, seguint una sèrie de pecularietats imposades pel protocol que aquesta eina de comunicació haurà de seguir.

A continuació farem un breu resum fent ús d'esquemes i diagrames que permetin situar al lector de les funcionalitats que la implementació d'aquesta eina ha de seguir, per tal de la seva correcta avaluació.

### 1.1.1 Accions de l'usuari

Un cop s'ha iniciat el *servidor*, caldrà connectar el primer client, aquest, portarà a terme una configuració amb el servidor; aquest procediment es pot veure de forma esquemàtica en la figura 1.

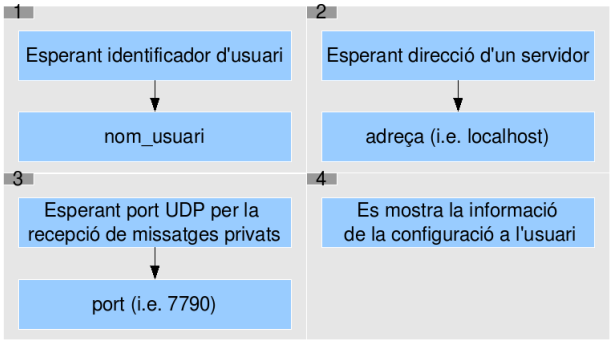


Figura 1: Procés de configuració d'un client.

Acabat el procés de connexió, configuració i enregistrament del client, ja podem interactuar amb aquest, és a dir, enviar missatges, demanar ajuda, sortir del xat ...etc. Totes aquestes operacions, queden reflexades en l'esquema de la figura 2.

### 1.1.2 Protocol client-servidor

En la següent llista enumerada veurem els detalls del protocol que s'ha d'implementar entre un client i un servidor; aquest consta de les següents característiques:

1. El servidor espera connexions en el port *TCP/8642*.
2. El client passarà de forma iterativa per les següents etapes:
   (a) Estat de configuració (veure secció 1.1.1).
   (b) Estat de connexió; el client es connecta al servidor per el port *TCP/8642* i el servidor espera el missatge d'identificació. Si es produeix algun error el client finalitza la sessió.
   (c) Estat d'identificació; consta de:
      i. El client envia un missatge d'identificació i espera el missatge *OK* del servidor.
      ii. El servidor envia el missatge *OK* i entra en espera del missatge de registre.
   (d) Estat de registre; consta de:

[1] Xat (català:conversa), que també es coneix amb el nom de conversa cibernètica, és un anglicisme que usualment es refereix a una comunicació escrita a través d'internet entre dues o més persones que es realitza de forma instantània.

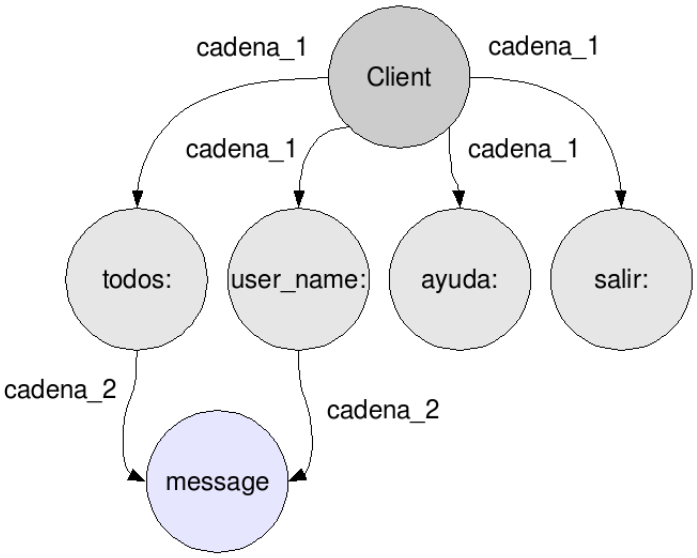


Figura 2: Comandes disponibles a executar per un client connectat a un servidor.

i. El client envia un missatge de registre.

ii. El servidor dóna d'alta a l'usuari , (desant l'identificador del mateix, l'adreça *IP* des de la qual es connecta i el port *UDP* d'on rebrà els missatges privats), envia un missatge *OK* i entra en un bucle d'espera de peticions.

Registrar-se amb un identificador d'usuari present actualment en el sistema, es considera un error. Quan s'acaba la connexió amb un client pel motiu que sigui, aquest es donarà de baixa en el registre.

(e) Estat de peticions al servidor; consta de:

i. Missatge pregunta, el client demana l'adreça *IP* d'un usuari i el port *UDP* pel qual es rebran els missatges provats. El servidor respon amb un missatge resposta. Si l'usuari no existeix el servidor enviarà com adreça *IP* "*null*" i el port *UDP* 0. Nota: el client iniciarà pel seu compte una comunicació amb el client en el port *UDP* i adreça *IP* obtingudes.

ii. Missatge de difusió, es demana al servidor que envïi un missatge a tots els clients connectats actualment. El servidor envia a tots els usuaris connectats (excepte al de la font del missatge de difusió) i envia un missatge *OK* a l'emisor un cop satisfeta la petició.

iii. Missatge de sortir, s'informa al servidor de l'eixida de la sessió.

Observacions generals:

1. Si no es cumpleix el protocol per part del client, ja sigui perque el missatge enviat és sintàcticament o semànticament incorrecte, el servidor envia un missatge *ERROR* al client i finalitza la sessió amb el mateix.

2. Si no es cumpleix el protocol per part del servidor, ja sigui perque el missatge enviat és sintàcticament o semànticament incorrecte, el client envia un missatge *SALIR* al servidor i el client finalitza la sessió.

3. La recepció d'un missatge *ERROR* obliga al client a finalitzar la sessió.

4. L'enviament d'un missatge *SALIR* per part del client obliga a aquest a finalitzar la sessió.

5. Si el client detecta que el servidor ha caigut, finalitza la sessió.

6. Si el servidor detecta que el client ha caigut o bé rep un missatge *SALIR*, finalitza la sessió amb el client, això implica donar de baixa a l'usuari enregistrat.

7. Qualsevol altra situació no contemplada en aquest protocol tan en el servidor com en el client porta a finalitzar la sessió ja sigui en el servidor o en el client.

8. No hi ha *timeouts*.

### 1.1.3 Protocol client-client

Un client pot enviar un missatge privat a un altre client (usuari) després que el primer hagi obtingut l'adreça *IP* i el port *UDP* del segon, a través del servidor. Un cop obtinguda l'adreça, s'envia un missatge privat. Així doncs, els clients poden rebre missatges tant per la connexió establerta amb el servidor com pel port *UDP* que han indicat l'usuari.
Tots aquests missatges es mostraran a l'usuari i s'especificarà la font.
Observació: qualsevol recepció pel port *UDP* que no correspongui a un missatge privat, serà descartat automàticament.

### 1.1.4 Missatges

La sintaxis dels missatges que estan permesos en aquest protocol, es poden veure en l'esquema de la figura 3.

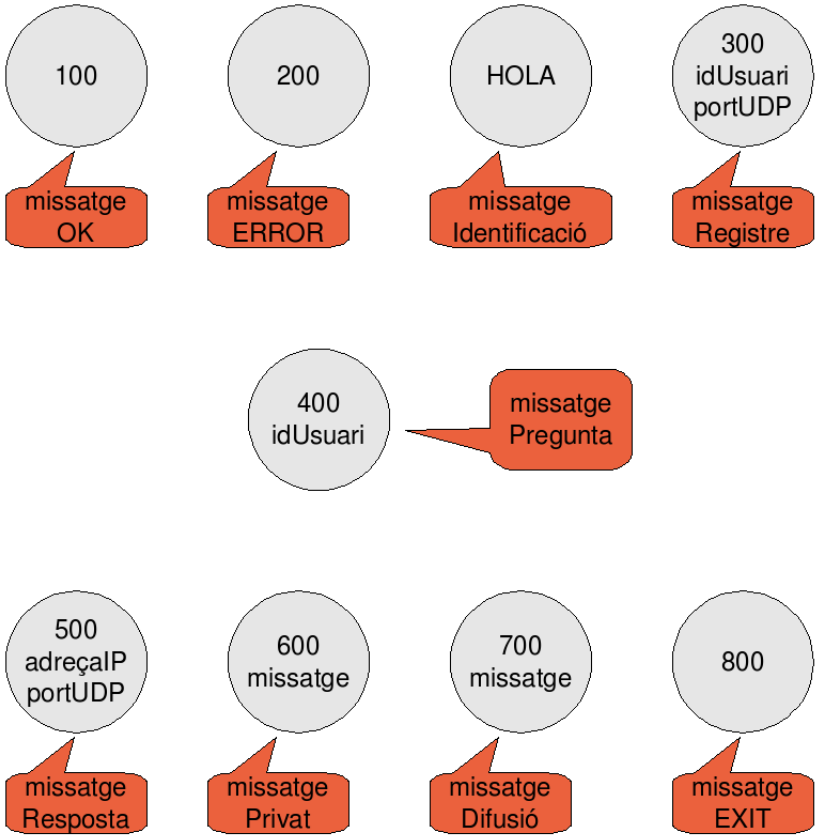


Figura 3: Missatges operatius en un client.

## 1.2 Metodologia de treball

En aquest apartat es fa un breu resum de la metodologia de treball que s'ha adoptat per resoldre la pràctica, és a dir, llenguatge de programació i eines usades pel desenvolupament de la pràctica.

### 1.2.1 Per què *python* i *c++* ?

Tal i com podrà apreciar el nostre *rewiever* la pràctica ha estat implementada usant dos llenguatges de programació: *c++* i *python*. Per què proposem dos dissenys? Doncs bé, lluny de la teoria que apunta que el fet de programar amb dos llenguatges diferents suposa incrementar exponencialment el volum de feina, nosaltres

proposem una metodologia basada en la implementació en *python* per la posterior "traducció" a *c++* tal i com es mostra en l'esquema de la figura *metod*.

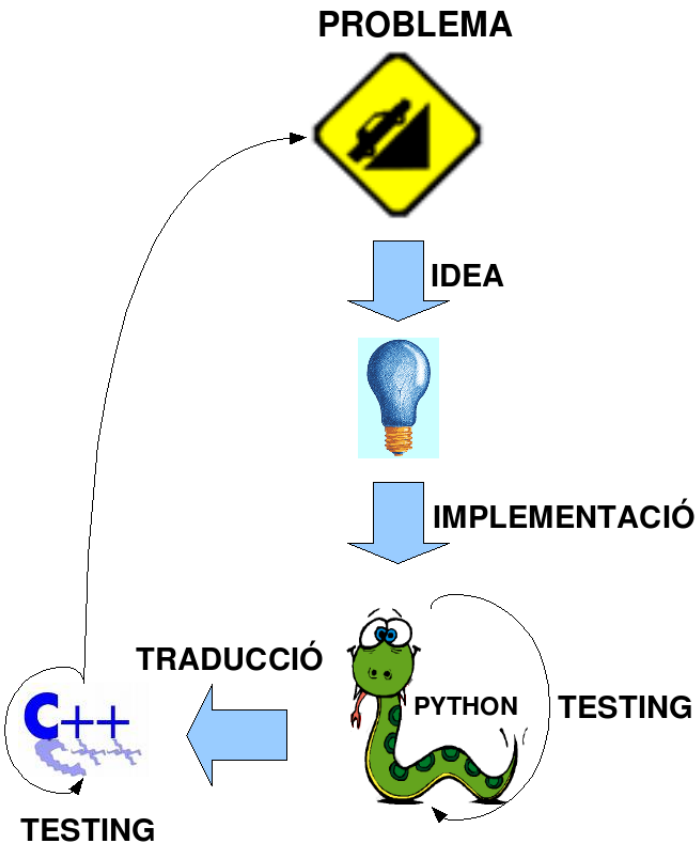


Figura 4: Mètode d'implementació.

El pas de traducció no es gens complicat, ja que l'estructura d'ambdós llenguatges és força semblant; a més a més el fet d'usar un disseny modular basat en classes facilita molt l'addesió poc a poc de les capes que formaran tota la implementació de la pràctica (en *python* i *c++*).

### 1.2.2 Eines usades

Aquesta pràctica s'ha desenvolupat en un entorn format per tres desenvolupadors (els tres integrants del grup). La necessitat de coordinar la feina entre nosaltres suposava un repte important, ja que una bona organització de grup facilitaria molt les coses i evitaria la realització de feina inútil, replicada ... o d'altres malentesos entre nosaltres.

Així doncs, per tal que tots nosaltres tinguéssim accés durant tot el dia al codi font (i d'altres arxius de la pràctica) hem creat un repositori *Subversion* amb 3 usuaris per tal que sigui el gestor de versions el que s'encarregui de resoldre els conflictes existents entre les versions que s'han portat a terme.

Pel que fa a l'eina de programació, cada integrant del grup ha usat el que li ha semblat més oportú, usant des de *Kwrite* a *Kate* o *Vim*.

La documentació del codi s'ha realitzat usant l'eina *Doxygen*[2] mentre que aquest informe s'ha desenvolupat usant el llenguatge LaTeX. Les imatges que s'han creat s'han dissenyat usant *oodraw* i el sistema operatiu que s'ha usat en el desenvolupament de la pràctica han estat les distribucions *Linux Debian* i *K-Ubuntu Edgy Eft*.

[2] *Doxygen* és un generador de documentació per *c++*, *c*, *java*, *objective-C*, *python* ... entre altres. Degut a la seva fàcil adaptació, funciona amb la majoria de sistemes *Unix*, així com en *Windows* i *Mac Os X*. La major part del codi està desenvolupada per *Dimitri van Heesch*.

# 2 Disseny de la pràctica

En aquest apartat, és comentaran els trets principals de les classes que formen tota la implementació en *python*. Un cop s'hagin repassat totes les classes d'aquesta implementació, es comentaran els trets diferenciadors entre ambdues versions.

## 2.1 Disseny de classes en *python*

En aquesta subsecció farem un repàs dels punts més importants del disseny efectuat amb *python*. L'ordre en que els diferents mòduls es van succeïnt en l'execució d'una connexió entre el servidor i un client ve esquematitzat en la figura 5.

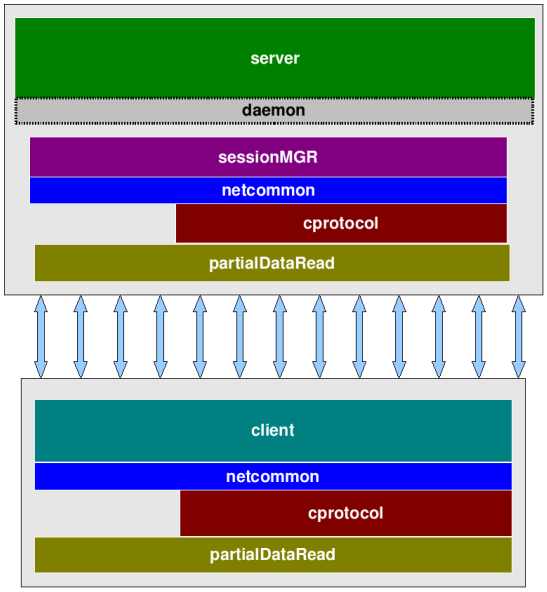


Figura 5: Diagrama de seqüència de les classes implementades en *python*.

Així doncs en aquesta figura es mostra que:

1. [S]: S'executa el mòdul *server*.
2. [S]: S'executa (o depèn de l'usuari) el mòdul *daemon*.
3. [C]: S'executa un client.
   Associat a l'execució del client tindrem que un seguit de mòduls es posaran en funcionament en ambdós parts, és a dir, en la banda del servidor *server* i del client *client*.
4. [S]: S'executa el mòdul *sessionMGR*.
5. [S]&[C]: S'executa el mòdul *netcommon*.
6. [S]&[C]: S'executa el mòdul *cprotocol*.
7. [S]&[C]: S'executa el mòdul *partialDataRead*.

On [S] significa que s'executa des del servidor, [C] que s'executa des del client i [S]&[C] que s'executa des d'ambdós punts.

Així doncs, entre l'esquema de la figura 5 i l'enumeració anterior, es pot arribar a entrendre que un disseny basat en la implementació de funcionalitats en classes, era la via òptima per afrontar de forma coherent i elegant la problemàtica de la pràctica, ja que l'ús d'un disseny modular ens permet que els diferents agregats o classes tinguin certa independència, la qual cosa ens habilita la reutilització de gran quantitat de codi i ens dóna com a resultat una aplicació molt robusta i coherent.

Un cop comentem el per què de l'existència de classes, anem a veure de forma molt breu els mòduls existents i la seva funcionalitat. Així doncs tindrem:

- Mòdul *server*: conté la classe *server*, encarregada de la realització de tasques per tal de realitzar serveis en benefici dels clients que es connectin a ell.
- Mòdul *daemon*: en aquest mòdul s'implementa la funció *daemon()* de *c++* que ens permet executar el servidor en mode *background* (o *dimoni*) en la nostra implementació en *python*, ja que en aquest llenguatge no existeix aquest mètode.
  Aquest mòdul és tracta d'una nova funcionalitat que s'ha afegit a la implementació de la pràctica.
- Mòdul *sessionMGR*: conté la classe *sessionMGR* i *clientSession* que són usades de forma exclusiva per el mòdul *server* per tal de gestionar (creació i destrucció) de les sessions dels clients (*clientSession*), és a dir, dels clients pròpiament dits i la informació associada: identificador, adreça *IP* ... etc.
- Mòdul *netcommon*: conté referències a la classe *selectInterface*. Aquestes referències són usades pels mòduls arrel *server* i *client* que els utilitzen per treballar amb els descriptors de fitxer dels diferents clients.
- Mòdul *cprotocol*: és el mòdul que conté la implementació de la gramàtica que han de seguir client i servidor. A més a més de definir els possibles estats d'un client, també és un mòdul que s'usa cada vegada que s'ha d'enviar i processar un missatge, és a dir, en les classes *client*, *server* i *clientSession* (es troba dins de *sessionMGR*).
- Mòdul *partialDataRead*: és la classe usada pel client i el servidor quan aquests es troben en estat d'espera de dades, és a dir, és la classe que s'encarrega de les lectures parcials.
- Mòdul *client*: conté la classe *client*, encarregada de realitzar aquelles tasques que ens permetran accedir als serveis que ens ofereix el servidor i així comunicar-nos amb d'altres usuaris (o clients) del servei.

## 2.2 Disseny de classes en *c++*

Seguint la mateixa metodologia i basant-se en el següent diagrama de seqüència de les classes que formen la implementació en *c++* (mostra't en la figura 6), comentarem en aquest cas, però, les diferències més significatives que hi ha entre la versió *python* i *c++*, ja que com hem comentat anteriorment segueixen la mateixa estructura.

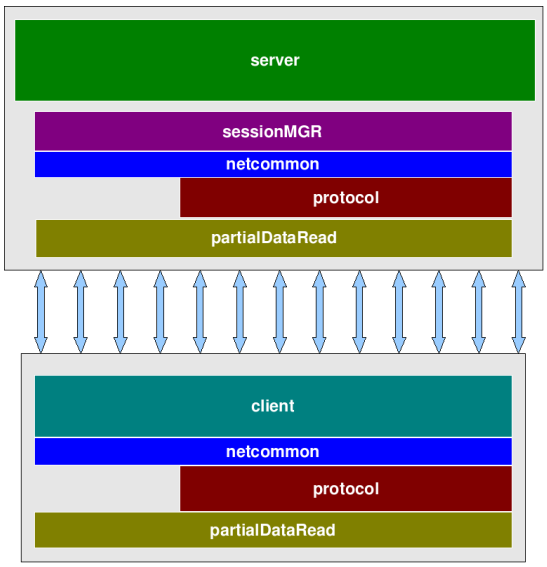


Figura 6: Diagrama de seqüència de les classes implementades en *c++*.

La diferència fonamental que volem destacar entre el disseny *python* i *c++* és la manera en què s'usa l'operació *select*.

En ambdues versions tota la funcionalitat del *select* es troba encapsulada dins de la classe *selectInterface* (dins de *netcommon*), que ens permet evitar errors ja que amb aquest encapsulament el programador no té la responsabilitat de comprovar si *fdmax* conté un valor correcte, de calcular-lo ... etc; ja que és la classe *selectInterface* la qui s'ocupa de fer-ho, reduïnt al màxim la possibilitat que aquest cometi un error. Dit en alt nivell, és una manera d'automatitzar el funcionament de *select* treient feina al desenvolupador.

Mentre en *python* aquest encapsulament treballa amb objectes la qual cosa és més elegant i polimòrfic, en la versió de *c++* es treballa directament amb els descriptors de fitxers la qual cosa resulta més farragós i menys abstracte.

## 2.3 Cas particular: lectures parcials

El punt que ens va comportar molts problemes, fou el referent a les lectures parcials, com podiem llegir dades incompletes i enviar-les i tenir en compte els separadors de línia.

A continuació comentarem dos punts molt importants que ens han permés posar remei a aquests problemes.

### 2.3.1 partialDataReader

Aquest mòdul ens va permetre llegir dades que no eren completes, és a dir, que eren parcials, acumular-les en un registre per tal d'unir-les amb la resta de dades que manquen per tractar-les tan bon punt es rebia el separador de línia.

### 2.3.2 Separador de línia

La problemàtica dels caràcters separadors de línia i els protocols d'internet, és el fet que a part dels missatges, també s'envien peticions que són de caracter protocolari. Així doncs, s'havia de controlar com marcavem el final de cada línia.

Usar el conegut "$\backslash n$" del *c++* era perillós en la implementació *python*. Així doncs vam decidir-nos per usar com a separador els caràcters "$\backslash r \backslash n$" ja que és la implementació usada de forma estàndard en tots els protocols on s'intercanvia text pla[3] (com en el nostre cas). Aquesta solució és més flexible del que sembla, ja que també es permet rebre dades amb separadors de tot tipus: "$\backslash n$", "$\backslash r$" i "$\backslash r \backslash n$", per tal d'acceptar aquelles implementacions que segons el nostre punt de vista no són del tot correctes.

## 2.4 Màquina d'estat del client

En aquesta secció es mostra de forma esquemàtica una màquina d'estats (emulant les màquines d'autòmats) on es pot veure els diferents estats en què un client es pot trobar, i com depenent dels missatges aquest client es trobarà dins d'un estat o dins d'un altre.

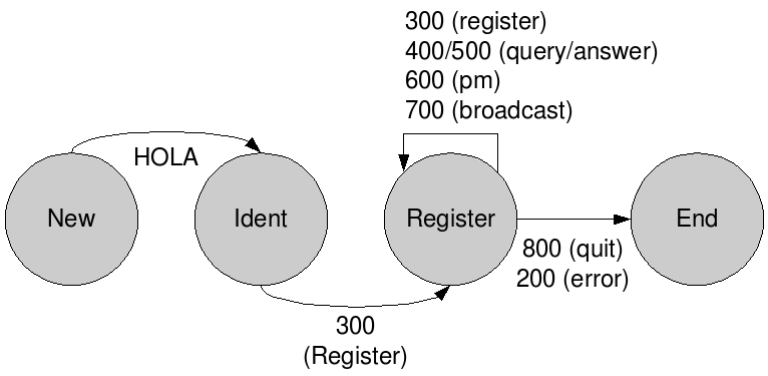


Figura 7: Màquina d'estats del client.

[3] Veure http://www.rfc-editor.org/EOLstory.txt.

## 2.5 Manuals d'execució

En aquest apartat mostrarem com s'han d'executar ambdues versions de la pràctica, és a dir, les crides que s'han de portar a terme, els arguments (si en tenen) ... etc, per tal que la pràctica funcioni correctament.

### 2.5.1 Versió *python*

Un cop estiguem ubicats dins de la carpeta del codi implementat en python */python*, haurem d'executar el servidor. Per fer-ho teclejarem:

```
./server.py
```

Aquesta seria una execució sense cap paràmetre, però el nostre servidor permet la parametrització de certs valors, que són:

- *-p <port>*; per tal d'establir un port *TCP* determinat (per defecte 8642).
- *-b <backlog>*; per tal d'establir el nombre de connexions en cua en mode espera (per defecte 10).
- *-D*; una nova funcionalitat que ens permet fer córrer el servidor en mode *background* o dimoni (per defecte desactivat).
- *-buf <size>*; per tal d'establir el tamany del buffer d'entrada (per defecte 4096).
- *-nw*; per activar l'opció d'ignorar els espais en blanc en els noms d'usuari (per defecte activat)
- *-bind <address>*; per tal d'establir adreces a connexió (per defecte s'accepten totes les adreces disponibles per totes les interfícies: "0").
- *-ctt*; per tal de netejar els caràcters *<IAC>*, *<TOF>* i *<OPT>* del protocol *telnet* (per defecte desactivat).
- *-e*; permet que hi hagi "*cmds*" buits des del client (per defecte desactivat).
- *–help*; per mostrar l'ajuda.

Un cop ja tenim el servidor en funcionament, és el moment d'executar un client, per fer-ho teclejarem dins de la metaixa carpeta */python* el següent:

```
./client.py
```

Aquesta seria una execució sense cap paràmetre, però els nostres clients permeten la parametrització de certs valors, que són:

- *nick@server:port*; nom_usuari@adreça_*IP*:port_*TCP*.
- *-p <port>*; per tal d'establir un port *UDP* determinat (per defecte assignat aleatòriament pel sistema operatiu).
- *-buf <size>*; per tal d'establir el tamany del buffer d'entrada (per defecte 4096).
- *-bind <address>*; per tal d'establir adreces a connexió (per defecte s'accepten totes les adreces disponibles per totes les interfícies: "0").
- *–help*; per mostrar l'ajuda.

### 2.5.2 Versió *c++*

Un cop estiguem ubicats en el directori del codi implementat en *c++*, caldrà compilar tots els fitxers, per fer-ho teclejarem:

```
make
```

Aquesta comanda cridarà al nostre *Makefile* i ens generarà els executables del client i servidor. Per executar el nostre servidor implementat en *c++* teclejarem:

```
./server
```

Aquesta seria una execució sense cap paràmetre, però el nostre servidor permet la parametrització de certs valors, que són:

- *-p <port>*; per tal d'establir un port *TCP* determinat (per defecte 8642).
- *-b <backlog>*; per tal d'establir el nombre de connexions en cua en mode espera (per defecte 10).
- *-D*; ens permet fer córrer el servidor en mode *background* o dimoni (per defecte desactivat).
- *–help*; per mostrar l'ajuda.

Un cop ja tenim el servidor en funcionament, és el moment d'executar un client, per fer-ho teclejarem dins de la meteixa carpeta el següent:

```
./client
```

Aquesta seria una execució sense cap paràmetre, però els nostres clients permeten la parametrització de certs valors, que són:

- *nick@server:port*; nom_usuari@adreça_*IP*:port_*TCP*.
- *-p <port>*; per tal d'establir un port *UDP* determinat (per defecte 0).
- *–help*; per mostrar l'ajuda.

# 3 Eina de testing

En aquesta secció explicarem com hem portat a terme les proves als nostres servidors.

## 3.1 testool.py

El testing està basat en un script implementat en *python* anomenat *testool.py*. Aquest script s'ha d'executar un cop tenim el servidor corrent, també és recomanable que hi hagi un client en funcionament que haurem de crear nosaltres mateixos i establir-li un port *UDP* inferior al 7000, per exemple el 6999.

Aquest script es llança amb la comanda:

```
./testtool.py argument1 argument2 argument3 argument4
```

On tenim que:

- El primer argument correspon al nom d'usuari que s'usarà per crear el client dins del test.
- El segon argument correspon a l'adreça *IP* que volem usar en la connexió.
- El tercer argument és el port *UDP* per el qual es volen rebre/enviar els missatge privats.
- El quart argument indica el nombre de missatges a enviar pel client.

Per tal d'entrendre millor els arguments anem a explicar que fa concretament aquest script:

El script crea un client amb identificador *argument1* que el que farà és enviar *argument4*x2 (ja que envia la cadena *"Eggs!!"* i la cadena *"Spam!!"* per separat) missatges de broadcast (*todos: missatge*) des de l'adreça *IP argument2* pel port *UDP argument3*. Un cop enviats tots els missatges, aquest client de testing enviarà el missatge *SALIR* i es desconnectarà del servidor.

## 3.2 test.sh

Una altra eina de test més exhaustiva és la que s'ha implementat en el *script test.sh*. Aquest *script* està basat amb l'anterior *script testool.py*.

Aquesta eina de testeig *test.sh* el que porta a terme és una crida mitjançant un bucle de 100 execucions de *testool.py*. Però, com s'ha resolt la crida del script inicial amb paràmetres? Doncs bé, fem un cop d'ull al codi:

```
#!/bin/bash
for i in ‘seq 1 100‘; do
    let port=7000+i
    python testtool.py bep$i localhost $port 10 &
done
```

Tal i com es pot veure és porta a terme un bucle de 100 iteracions on es crida a *testool.py* amb els arguments:

- Usuari *bep$i*; això significa que a cada volta de bucle, es crida a l'usuari *bep1*, *bep2*, *bep3* ...i així fins *bep100*.
- Adreça *IP localhost* en totes les iteracions.
- Port *UDP $port*, de la mateixa forma que en el cas dels usuaris, el port s'incrementa en una unitat per cada volta de bucle per tal d'evitar que la informació sigui molt caòtica a la resta de clients que ja formen part del xat. Es comença pel port 7000 i s'acabarà pel 7100.
- El nombre de missatges a enviar és 10, la qual cosa cada client enviarà 10 cops *Eggs!!* i 10 cops *Spam!!*.

# 4 Conclusions

En aquesta pràctica hem après diferents coses que han aportat molt a la nostra formació en aquesta assignatura, però també en tots els àmbits.

Hem après a treballar amb equip i a utilitzar aquelles eines que ens permeten fer-ho. Hem ampliat els nostres coneixements en la temàtica de la pràctica, és a dir, amb la utilització de *sockets*. També a criticar el treball fet mitjançant els tests que s'han creat per veure les possibilitats, capacitat i robustesa del nostre disseny. També s'ha realitzat un treball força interessant i molt sovint inexistent ens molts projectes de software com és la realització de documentació del mateix, en aquest cas en dos àmbits: codi font a mesura que aquest s'implementava i també de la documentació recopilatòria final (aquest informe) per tal d'expressar els trets generals de tot el que ha envoltat el desenvolupament d'aquesta pràctica.

Esperem també, més endavant, aprendre a realitzar avaluacions objectives de les propostes d'altri quan portem a terme el *peer review* i a comparar ambdós dissenys per tal de veure les mancances o avantatges tan del nostre disseny com el de l'altre grup.

En definitiva un conjunt d'aprenentatges que pensem que són molt interessants, ja que engloben totes les etapes que un bon desenvolupament de software ha de tenir, en el nostre cas, de software lliure.

# 5 Llicència del document