View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Aplicació de Twitter Descentralitzat

Projecte de Final de Carrera | Albert Montané Blanchart

Agraïments

No seria just entregar aquesta memòria sense deixar constància de totes les persones que, d'una manera o una altra, han fet que aquest projecte sigui possible.

En primer lloc m'agradaria agrair a en Joan Manel Marqués l'interès i la disposició que ha mostrat en tot moment per a donar-me un cop de mà, des del dia que ens vam reunir per a concretar el tema del projecte fins al dia en que aquesta memòria s'ha donat per acabada. També voldria agrair especialment a en Daniel Làzaro, coautor de CoDeS, la paciència que ha tingut amb mi i les bones idees que m'ha donat per a resoldre satisfactòriament un dels punts més complexos del projecte a nivell tècnic.

Una de les contribucions més importants a aquest projecte ha sigut la que ha fet la gent d'Abiquo Holdings S.L., en especial el Marc, el Xavi i el Salvador. Gràcies a la flexibilitat horària que m'han permès, aquest projecte s'ha pogut finalitzar dins els terminis. Sóc plenament conscient que, si jo hagués estat treballant en una altra empresa, segurament no se m'hagués donat aquest tracte de favor. És per això que vull agrair-los l'ajuda i el suport que m'han donat.

Voldria recordar-me també d'en Xavier Llinàs i de tota la classe de HDC. És possiblement una de les assignatures on més be m'ho he passat en tota la carrera, i alhora una de les que més m'han ensenyat en un dels camps que més fluixos porto: les presentacions orals i escrites. Gràcies a ells, aquest treball és molt millor del que podria haver sigut!

També m'agradaria donar les gràcies a totes aquelles persones que s'han interessat per l'estat del projecte i m'han donat ànims durant aquests llargs sis mesos. En especial, voldria recordar-me del Raul i de l'Abel, que m'han ofert de bon grat els seus consells des de l'experiència d' algú que ja ha fet un Projecte de Final de Carrera.

Finalment, voldria agrair a la meva família els ànims que m'han donat i la paciència que han tingut amb mi, ja no tan sols durant el procés d'elaboració d'aquest projecte, sinó durant tot el temps que he estat a la Facultat d'Informàtica de Barcelona. I, sobretot, voldria donar les gràcies a la Montse per tot el que ha fet per mi aquests dies. Sense la seva comprensió, res d'això hagués sigut possible.

Moltes gràcies a tots!

Índex

Introducció 6
Definició del projecte 6
Motivació 7
Estructura de la memòria 7
HorNet: Vista general 9
Models de desplegament 9
Contenidor local 9
Contenidor remot 10
Híbrid 11
Tecnologia utilitzada 12
Client 12
Contenidor 12
Servidor 12
Disseny de l'aplicació 13
Funcionalitats bàsiques 13
Publicació de missatges 13
Subscripció a usuaris 13
Model de dades bàsic 13
Funcionalitats complementàries 14
Perfils 14
Mencions 15
Retransmissió 16
Cerca 17
Eliminació de missatges 17
Consideracions de disseny 18
Identificador d'Usuari 18
Representació única de Missatges 19
Model de dades 19
Arquitectura orientada a serveis 21

Estructuració de la informació 21
Model per a HorNet: model completament distribuït 24
Servei User.Messages 24
Servei User.Timeline 25
Servei User.Followers 26
Servei User.Following 27
Servei User.Mentions 27
Servei User.Profile 28
Consideracions sobre els serveis 29
Servei Authentication 30
Servei Indexation 31
Especificació del client 32
Introducció: Servlets en Jetty 32
Servlets 32
La classe JettyLauncher 33
Enllaços vs Redireccions vs Formularis 34
Especificació dels Servlets de HorNet 35
LoginServlet 35
RegisterServlet 35
MainServlet 36
SendTweetServlet 36
ReceivedMessagesServlet 37
SearchServlet 37
SelfProfileServlet 37
ViewProfileServlet 38
FollowServlet 38
Diagrama de navegabilitat 38
La connexió client-servidor 39
La interfície DataManager 40
La classe CoDeSDataManager 41

Altres components del client 41
Classe Cache 42
Arxiu style.css 43
Arxiu ProfileTemplate.xml 43
Especificació del servidor 45
CoDeS 45
Overview dels serveis de CoDeS 47
Comunicació amb els serveis 50
Opció 1: Enviament de missatge a través de l'API 50
Opció 2: Apertura de socket en el desplegament 51
Opció 3: Missatge + Socket 52
Classe PortPool 53
Classe DeliveryMessage 54
Classe ServiceInvokingApp 55
Especificació dels serveis 56
Servei User.Messages 57
Servei User.Timeline 58
Servei User.Followers 60
Servei User.Following 62
Servei User.Mentions 63
Servei User.Profile 65
Servei Authentication 67
Servei Indexation 68
Persistència de les dades 69
Persistència en memòria 70
Persisténcia en disc 70
Metodologia 72
Planificació 72
Anàlisi de costos 74
Conclusions 76

Valoració personal 77
Referències 78
Bibliografia 79
Annex: Instruccions de desplegament de HorNet 81
HorNet.tar.gz 81
Precondicions d'execució 81
Execució dels components 82
Servidor 82
Contenidor 82

Introducció

No es pot negar que les últimes dècades han sigut d'una importància tecnològica elevadíssima. Però d'entre tots els nombrosos avenços que hem pogut gaudir, el més important ha sigut sens dubte l'aparició d'Internet. En els gairebé cinquanta anys que han passat des que tenim accés a la xarxa, hem pogut viure una veritable revolució en la manera com es transmet i divulga la informació. Aquesta facilitat de comunicació ha potenciat l'aparició de les xarxes socials: una sèrie d'aplicacions que han subministrat a les persones eines per a comunicar-se entre elles gairebé lliurement. És així com han nascut portals tan coneguts com *Facebook*[1] o *Twitter*[2].

Tanmateix, tot i les bondats d'aquestes tecnologies de comunicació, cal reconèixer que també té limitacions. Els usuaris, per tal de poder coordinar-se, accedeixen a un servidor central. A més, per a poder subministrar un servei sense interrupcions no n'hi ha prou amb un sol servidor, donat que aquest podria fallar, i requerir un determinat nombre de màquines dedicades pot tornar-se un problema per a algú que, potser, simplement vol un servidor web per a desplegar una aplicació.

És per això que projectes com aquest tenen sentit: per a trobar solucions escalables a problemes que, cada cop més, aniran apareixent en una societat com la nostra en la que l'accés a la informació és, si bé potser no una necessitat, una cosa quotidiana.

Definició del projecte

Agafant com a referència una aplicació de complexitat relativament baixa, com és en el nostre cas *Twitter*, mirar de re dissenyar-la per tal que estigui formada per un bon nombre de components de la granularitat més fina possible, per tal que desplegar-la en un entorn completament descentralitzat permeti un millor aprofitament dels recursos.

[1] *Facebook*: És un dels portals d'internet més coneguts actualment, tot un exemple de xarxa social pura. Els usuaris poden publicar tot tipus d'informació (text, signatures, fotografies, invitacions a events) i disposen d'un elaborat perfil mitjançant el qual els altres usuaris els poden localitzar.

[2] *Twitter:* xarxa social orientada principalment a la missatgeria. Els usuaris disposen d'un sistema de publicació-subscripció i d'un rudimentari sistema de perfils

Hem decidit batejar aquesta aplicació com a *HorNet*, que és la traducció a l'anglès d'un tipus de vespa que viu en comunitats grans. Creiem que és un nom adient, donat que *HorNet* és una implementació d'una xarxa social, d'usuaris que es comuniquen mitjançant missatges curts de text.

Motivació

Creiem que la necessitat de recursos de software, avui en dia, és una realitat. Cada cop més gent disposa de dispositius electrònics amb accés a la xarxa, i per tant cada cop hi ha més recursos dedicats a donar servei a aquestes màquines. Un dels principals problemes de la informàtica actual és com proporcionar aquests recursos de manera més fàcil i amb un cost més baix. Per això una de les tendències actuals de la informàtica es basa en el desenvolupament de tècniques que permetin una obtenció més senzilla dels tan necessitats recursos. És el que es coneix com a *Cloud Computing*[3].

Per altra banda, se'ns ha presentat la oportunitat de contribuir en un projecte de codi lliure desenvolupats per universitaris catalans. Aquest projecte, en el que s'ha basat l'aplicació que desenvoluparem, consisteix en una infraestructura dissenyada precisament per a respondre a la cada cop més elevada necessitat de serveis software. Es tracta d'un projecte en desenvolupament, que funciona correctament però que encara no disposa de massa serveis diferents per a oferir als seus usuaris. Mitjançant aquest treball, estaríem contribuint a que el projecte *CoDeS*[4], del que parlarem més endavant, sigui una mica més ric.

Resumint: ens trobem davant d'una tendència en alça, i creiem que desenvolupar una aplicació d'aquestes característiques pot resultar en una bona oportunitat per a aprendre a desenvolupar un tipus de productes que, cada cop més, aniran sent productes informàtics quotidians.

Estructura de la memòria

En aquest document, descriurem el procés d'elaboració de l'aplicació *HorNet*. L'hem estructurat d'una manera gairebé cronològica, seguint l'ordre en el que s'han anat aplicant les fases de desenvolupament. Els blocs de contingut del document són els següents:

---

[3] *Cloud computing:* Paradigma de comunicació en xarxa en el qual es destinen uns recursos, organitzats en una determinada infraestructura, per a facilitar serveis software als usuaris sota demanda.

[4] *CoDeS*: projecte de codi lliure desenvolupat per estudiants universitaris de Catalunya, que permet desplegar un núvol sense recursos dedicats a tal efecte.

- **Vista general**: fa una ullada externa a l'estructura de l'aplicació: quins components té, quines tecnologies utilitza per a la seva implementació, quins models de comunicació defineix...
- **Disseny**: dona un punt de vista més detallat de les funcionalitats de l'aplicació, els tipus de dades que fa servir, els protocols simples de comunicació...
- **Especificació del client**: dóna un punt de vista encara més detallat que en la secció de disseny sobre un dels dos grans components de l'aplicació: la capa de client. En aquesta secció veurem en detall com els punts de l'apartat anterior (disseny) es veuen afectats quan s'apliquen sobre la tecnologia que els implementarà.
- **Especificació del servidors**: de forma anàloga a l'apartat anterior, en aquest bloc de contingut veurem com queda el disseny de la capa de servidor quan hi apliquem la tecnologia per a la que ha estat dissenyat.
- **Metodologia**: en aquest últim bloc de contingut podrem veure amb detall quin ha sigut el procés de desenvolupament de l'aplicació, quina ha sigut la seva planificació, i quin seria l'anàlisi de costos.

HorNet: Vista general

L'aplicació que volem descriure en aquest document s'anomena *HorNet*. Com hom es pot imaginar pel títol del document, es tracta d'una aplicació de Twitter, dissenyada per funcionar en un entorn descentralitzat.

El funcionament teòric es basa en disposar de l'accés a un grup d'ordinadors (un núvol o *cloud*) que donen servei a l'aplicació. D'aquí ve la seva naturalesa descentralitzada. Tanmateix, tot i tractar-se d'un paradigma diferent al clàssic client-servidor, alguns components son comuns, i d'altres es poden abstraure i agrupar per tal de fer el comportament més entenedor. Així, podem diferenciar entre aquestes tres parts:

- Client web: és l'encarregat de mostrar una *GUI*[5] entenedora l'usuari, d'interpretar-ne les seves ordres, de transmetre-les al contenidor web i d'interpretar els resultats que aquest li enviï.
- Contenidor web: s'encarrega de fer d'enllaç entre el client i el servidor, mitjançant protocol HTTP.
- Servidor: en el cas d'aquest model descentralitzat, és una figura abstracta que representa tota la informació i recursos destinats a HorNet dins del núvol. Quan li arriba una petició del contenidor web, la redirigeix al node que li toca per tal que aquest la pugui atendre i respondre.

Cada un d'aquests components es pot executar en un o més nodes, segons el model de desplegament que haguem escollit.

Models de desplegament

El lloc on ubiquem cada un dels components determinarà quin és el model de desplegament.

Contenidor Local

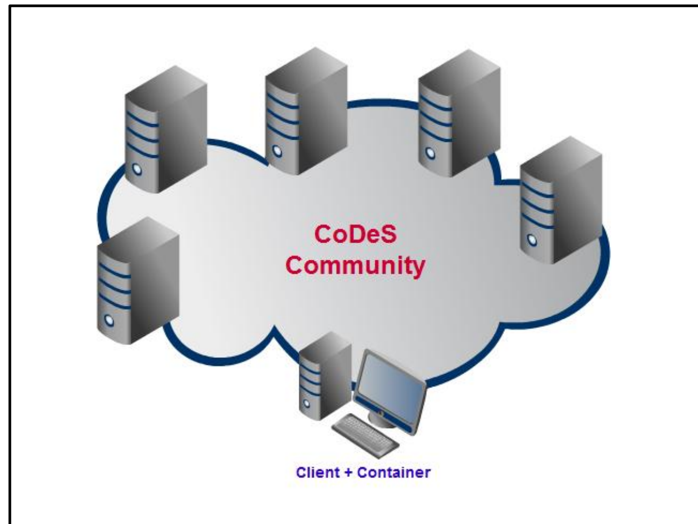
El model més típic és el de contenidor local. En aquest model, el client web i el contenidor web es despleguen en la mateixa màquina, i el client accedeix a l'aplicació com a localhost[6]. Per tal que el contenidor web es pugui comunicar amb el servidor,

---

[5] GUI: Sigles de Graphical User Interface, literalment, interfície gràfica d'usuari. És un component software que facilita la comunicació entre usuari i aplicació.

[6] localhost: adreça de xarxa que fa referéncia al propi ordinador

cal que aquest es trobi dins del núvol. Així doncs, aquest model de desplegament té una limitació, i és que l'usuari client de l'aplicació s'ha de trobar dins del núvol.



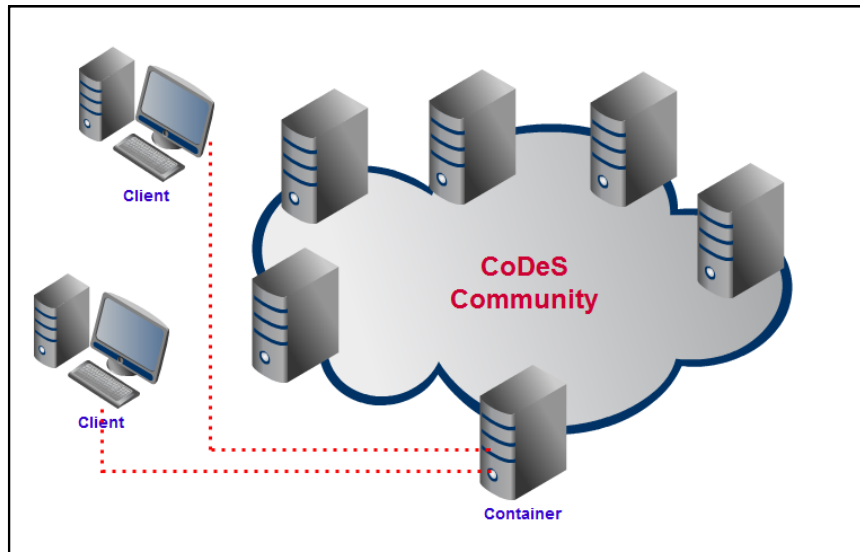
Desplegament amb contenidor local (font: pròpia)

Contenidor remot

Un altre model vàlid seria el model de contenidor remot. En aquest model, el client web i el contenidor web es despleguen en una màquina diferent, i el client necessita saber l'adreça del contenidor per a poder-s'hi comunicar. A diferència de l'anterior model, ja no cal que el client estigui ubicat dins del núvol: pot comunicar-se des de fora sempre i quant conegui l'adreça del contenidor.

Aquest model de desplegament té un inconvenient respecte a l'anterior, i és que la comunicació client-contenidor no només és remota sinó que ja no està limitada a un sol client per contenidor: qualsevol client que vulgui fer servir l'aplicació necessitarà comunicar-se amb el contenidor publicat, i això és un potencial coll d'ampolla. Tanmateix, es pot minimitzar aquest efecte replicant el contenidor i utilitzant tècniques de *load-balancing*[7].

[7] *Load-balancing*: tècniques utilitzades per a distribuir la càrrega entre diversos nodes, ja sigui per una millor *performance* o per a garantir una alta disponibilitat.

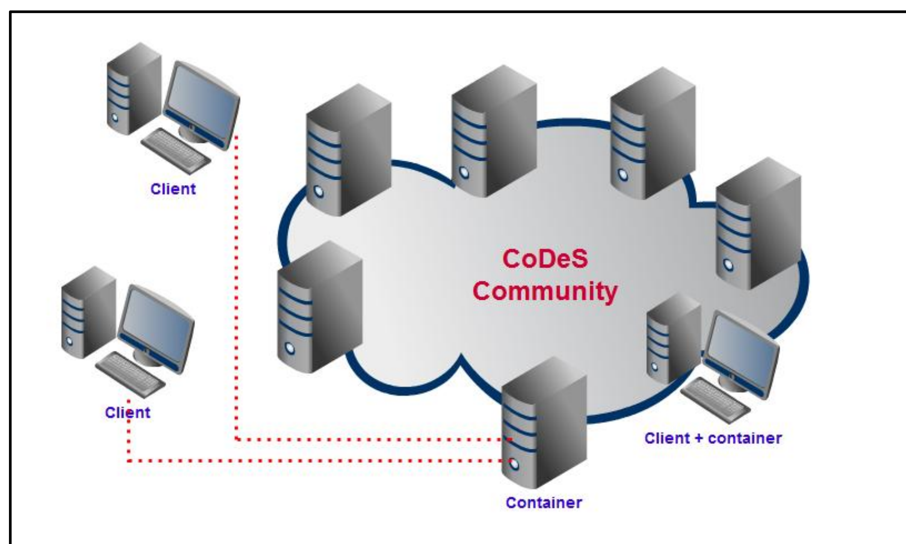


Desplegament amb contenidor remot (font: pròpia)

Híbrid

Finalment, es poden combinar els dos models de desplegament anteriors per a obtenir un model mixt en el qual es farà servir el contenidor que sigui millor segons les circumstàncies.

Els clients que es troben dins del núvol poden utilitzar el contenidor local que té un temps de resposta més ràpid, i els clients que es troben fora del núvol poden utilitzar un dels contenidors replicats per la xarxa.



Desplegament híbrid (font: pròpia)

Tecnologia utilitzada

Les tecnologies que s'han utilitzat per a cada un dels tres components són les que es descriuen a continuació.

Client

- HTML[8]: El client que fem servir és un client web, i per tant la majoria del que l'usuari veu és codi HTML generat des del contenidor mitjançant Java Servlets.
- AJAX[9]: Algunes peticions del client al contenidor es fan amb peticions HTTP[10] normals, però algunes, les que es realitzen més sovint, es fan asíncronament mitjançant AJAX per no recarregar la pàgina sencera cada cop.

Contenidor

- Java[11] Servlets[12]: La interpretació de les peticions i la generació dels continguts que es mostren al client es fa mitjançant aquesta tecnologia.
- Jetty[13]: Hem escollit Jetty com a contenidor perquè ens proporciona la potència necessària per al tipus de client que hem escollit, ens facilita suport a Java Servlets i a més és de codi lliure.

Servidor

- CoDeS: El servidor utilitza CoDeS, un projecte de codi lliure implementat en Java que implementa un SOA[14].
- Java: El codi de comunicació entre el contenidor i el servidor està implementat en Java. El fet que tan CoDeS com el contenidor facin servir Java ens facilita molt la comunicació entre components.

---

[8] HTML: Sigles de HyperText Markup Language, el llenguatge més comú utilitzat per a la interpretació de pàgines web.

[9] AJAX: És el nom comercial que rep *XMLHTTPRequest*, una classe de JavaScript que permet realitzar peticions HTTP asíncronament sense recarregar tota la pàgina.

[10] HTTP: Sigles de *HyperText Transfer Protocol*, el protocol bàsic de comunicació que es fa servir en la comunicació web.

[11] Java: llenguatge de programació extensivament utilitzat degut, entre altres coses, al seu suport multiplataforma.

[12] Servlets: Petites aplicacions de java que permeten interpretar peticions HTTP.

[13] Jetty: Contenidor de codi lliure 100% basat en java, desenvolupat per Eclipse Foundation.

[14] SOA: Sigles de *Service Oriented Architecture*, arquitectura orientada a serveis.

Disseny de l'aplicació

Volem dissenyar una aplicació similar a *Twitter*. En una aplicació d'aquestes característiques hi ha una sèrie de funcionalitats bàsiques que no ens podem saltar (o altrament estaríem parlant d'una aplicació completament diferent) i una sèrie de funcionalitats complementàries que fan que l'aplicació sigui més rica però sense les quals l'aplicació podria funcionar correctament.

Funcionalitats bàsiques

Hem decidit que només compten com a bàsiques per a la nostra aplicació dues de les funcionalitats de *Twitter*: la **publicació** i la **subscripció**.

Publicació de missatges

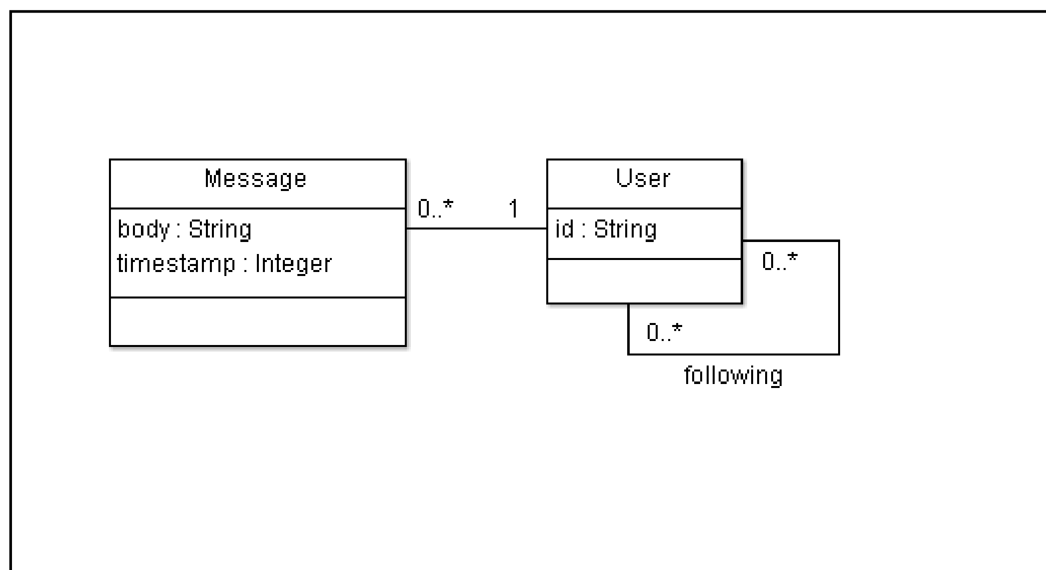
Mitjançant la nostra aplicació, els usuaris que s'han registrat poden publicar notícies. A *Twitter*, aquestes notícies son missatges de text d'una longitud de com a molt 140 caràcters. Una notícia s'associa a un i a només un autor, i es retransmet cap a tots els seguidors d'aquest autor tot i que són visibles potencialment per tots els membres de la comunitat.

Subscripció a usuaris

La funció de subscripció és la que permet que un usuari es subscrigui a un altre usuari per a rebre les notícies que aquest últim publica. La relació de subscripció es unidireccional: que un usuari A es subscrigui a un usuari B significa que A llegirà les notícies de B, però l'usuari B no llegirà les notícies de l'usuari A a no ser que B es subscrigui a A.

Model de dades bàsic

Amb les funcionalitats bàsiques, la representació del model de dades de l'aplicació és el següent:



Model de dades simplificat (font: pròpia)

Estaríem parlant de dues classes: la classe *User*, que representaria l'usuari de la comunitat, i la classe *Message*, que representa els missatges que aquests usuaris publiquen i llegeixen.

Pel que fa a les relacions entre classes, cada una d'elles representa una funcionalitat de l'aplicació. La relació *User-User*, amb una cardinalitat de *-*, representa la funcionalitat de subscripció (un usuari pot estar subscrit a zero o més usuaris, i pot tenir zero o més usuaris subscrits). La relació *Message-User*, amb una cardinalitat de *-1, representa la funcionalitat de publicació (un usuari pot publicar zero o més missatges, i aquests missatges tenen un i només un usuari per autor).

Funcionalitats complementàries

D'entre les moltes funcionalitats de Twitter, n'hem escollit unes quantes com a funcionalitats complementàries de cara a preparar el disseny de la nostra aplicació: **perfils**, **mencions**, **retransmissions**, **cerques** i **eliminació de missatges**. L'aplicació de les funcionalitats addicionals és modular, el qual significa que es poden aplicar independentment sense requerir-se les unes a les altres.

Perfils

En una aplicació social d'aquestes característiques sempre és desitjable desar informació personal per tal que els contactes puguin saber més coses sobre nosaltres. Tanmateix, amb el model actual, la única informació que està associada directament a

l'usuari és el seu identificador. Per tal de poder emmagatzemar més informació sobre els usuaris, cal modificar el model. Necessitem que aquesta informació addicional s'emmagatzemi en alguna banda.

El model de dades no es veu afectat pel que fa a les relacions entre classes. La única diferència és que canviarà la manera de veure la classe User.

Si afegim perfils, ja no caldrà que identifiquem un usuari pel seu nom: podrem assignar-li un identificador numèric, transparent a la interfície, i incloure el nom d'usuari visible com a una part del perfil. Això ens obrirà noves portes, com per exemple la possibilitat de canviar de nom d'usuari si ens en cansem.

Una altra possibilitat que ens facilita aquest canvi d'identificador, és la d'aplicar multi perfils a la plataforma, és a dir, permetre tenir diversos perfils creats per a fer-los servir segons el tipus de missatge que vulguem enviar, o el tipus de públic al que ens vulguem dirigir.

En el tipus de perfil que hem escollit per a la nostra aplicació cal diferenciar entre dos tipus de camps:

- Camps obligatoris: són els que es necessiten per a que funcioni l'aplicació correctament. Aquests camps són:
  - Identificador numèric
  - Nom d'usuari
  - Contrasenya
  - Avatar (imatge)
- Camps opcionals: l'usuari pot escollir d'omplir-los o no, i no estan definits formalment en el codi de l'aplicació. La definició d'aquests camps es troba en un arxiu de configuració del contenidor, que indica a l'aplicació web quins camps ha de reconèixer. S'emmagatzemen al model de dades mitjançant un diccionari.

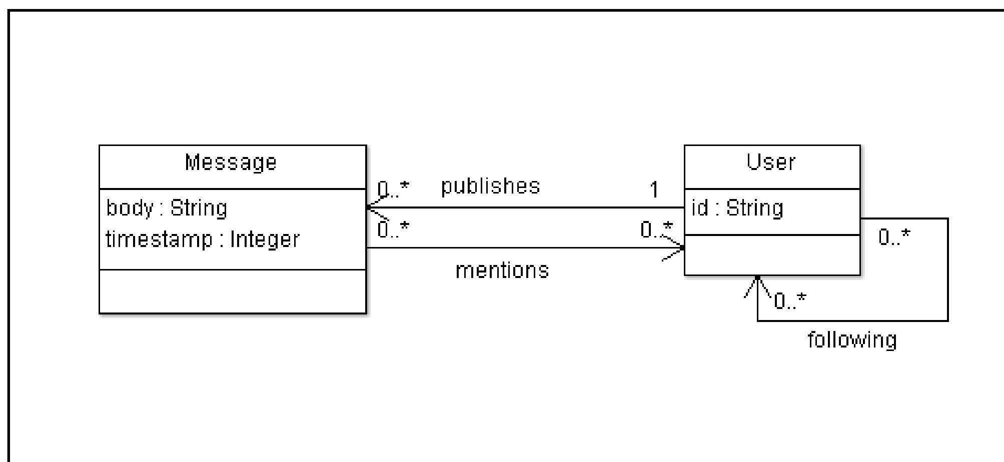
Mencions

Una bona manera d'enllaçar amb membres de la comunitat que no coneixes o que no et coneixen a tu, és a través del sistema de mencions. En un missatge en el qual s'hagi indicat degudament, és possible mencionar un usuari de la comunitat. Aquest missatge serà públic, igual que un missatge normal i corrent, però tindrà la peculiaritat

que l'usuari que hem mencionat veurà el missatge a la seva llista, independentment de si es seguidor nostre o no.

El sistema que s'utilitza a Twitter per a indicar que estem mencionant un usuari és afegint una @ al davant del seu identificador. Així, si nosaltres escrivim: "@User2 Hola, què tal?", l'usuari amb identificador User2 serà mencionat, i veurà el nostre missatge independentment de si és seguidor nostre o no.

Si afegim aquesta funcionalitat, cal que tinguem en compte que ara els missatges ja no només estaran associats a un usuari, el seu autor. Ara a més podran estar associades a diversos usuaris, en cas que se'ls mencioni explícitament. Estaríem parlant d'un model de dades com aquest:



Model de dades amb funcionalitats bàsiques (font: pròpia)

La relació *publishes* segueix existint i és obligatòria ja que té cardinalitat 1 en l'extrem de l'usuari. En canvi, ara apareix una nova relació amb cardinalitat *-* que relacionarà un missatge amb els usuaris que hi apareguin mencionat. Cada usuari, doncs, pot estar mencionat a zero o més missatges, i un missatge pot mencionar a zero o més usuaris.

Retransmissió

Una altra funcionalitat típica del *twitter*, es la possibilitat de "retwittejar" un missatge que ha publicat algú. D'aquesta manera, tots els nostres seguidors podrien veure el missatge, tal i com si l'haguéssim escrit nosaltres, amb la única diferència que com a autor en figuraria el seu autor inicial, tal i com si els nostres seguidors fossin els seus, però aplicat només al missatge al que hem fet *retweet*.

A efectes pràctics, un missatge retransmès és com un missatge nou, publicat per nosaltres: la hora que es mostra en el cos del missatge és la hora en la que el missatge s'ha escrit, però la posició en la que el missatge s'ordena és la hora en la que el missatge s'ha retransmès, en comptes de la hora en la que s'ha escrit el missatge original. Per tant una bona manera d'implementar la retransmissió és considerant-lo un missatge propi, i afegint una marca que indiqui que el seu autor original no som nosaltres, per tal que l'aplicació l'interpreti com una retransmissió.

Cerca

Una de les eines més potents per a facilitar que usuaris que es coneixen i no saben que es troben en la mateixa xarxa social puguin trobar-se i seguir-se, és sens dubte un cercador. Sabent una dada d'aquell usuari, com per exemple el seu nom de pila o a quina empresa treballa, se'l pugui reconèixer al sistema.

Un dels trets importants que volem que tingui el nostre cercador és una bona disponibilitat de cara a la privacitat. Volem que sigui cada usuari el que decideixi quina informació vol que sigui pública al sistema i quina vol que sigui privada.

Decisions sobre l'abast del cercador

A l'hora de decidir el disseny de la funció de cerca, cal considerar a quines dades volem que aquest tingui accés. Tenim principalment dues opcions, no exclusives entre elles:

- **Perfil d'usuari**: Totes les dades, o una part, del perfil dels usuaris son subjectives de ser cercades en un moment donat. Si volem que el nostre cercador tingui accés a aquestes dades, tindrem la capacitat de trobar usuaris a través de dades personals, empresarials, acadèmiques...
- **Missatges publicats**: Una facultat que pot afegir potència al cercador és la possibilitat de buscar entre els missatges que un usuari ha publicat. Això també ens pot permetre, a més de la intenció inicial de la funcionalitat que és la de localitzar usuaris més fàcilment, cercar i trobar notícies interessants, de cara a retransmetre-les o simplement a visualitzar-les.

Eliminació de missatges

En un sistema de publicació-subscripció d'aquestes característiques, un dels punts crítics sempre és l'aplicació de les operacions de modificació del model de dades. En el cas concret del missatges, cal tenir en compte que cada un d'ells es

publicarà a diversos usuaris, i que per tant podria tenir diferents representacions en cada sistema.

En el cas de les modificacions, pot ser molt complicat mantenir la consistència del model de dades en cas que els missatges tinguin moltes representacions. Tanmateix, en el cas de les baixes de missatge, no té per què ser així. Si aconseguim que els missatges es puguin identificar independentment de quantes vegades estiguin representats, podem assegurar que si un missatge existeix, el seu contingut és correcte. No podem dir el mateix de les modificacions. És per això que volem permetre que els missatges puguin ser, com a mínim, eliminats del sistema. En un cas funcional, es podria "modificar" un missatge eliminant-lo i tornant-lo a publicar amb un identificador nou.

Consideracions de disseny

A més de les funcionalitats en si, cal tenir en compte una sèrie d'aspectes importants respecte al model de dades, que poden afectar directament a la manera com l'especifiquem i implementem.

Identificador d'Usuari

En *Twitter* i aplicacions per l'estil, els usuaris gairebé sempre coneixen el seu identificador. Per dir-ho d'alguna manera, el nom d'usuari és un identificador informal, ens permet saber qui es un usuari, i que es pugui autenticar amb el seu nom d'usuari conegut sense necessitar recordar cap altre identificador. Tanmateix, des del punt de vista de l'aplicació, això no ha de ser necessàriament així. És possible que l'aplicació identifiqui els usuaris amb un identificador numèric que ni tan sols coneixen.

És per això que cal que decidim com volem que els usuaris s'identifiquin a l'aplicació, si ho fem mitjançant un identificador formal, i que el nom d'usuari es mantingui com un camp més del perfil, o bé si ho fem mitjançant un identificador informal, i forçar als usuaris a que no el puguin canviar.

En el cas de *HorNet*, hem decidit identificar els usuaris amb un identificador formal numèric, un nombre natural superior a zero. Aquest identificador serà "1" per al primer usuari que es registri al sistema i anirà incrementant per a cada usuari que es registri després. L'usuari no coneixerà aquest identificador, així que cal mantenir el nom d'usuari com a identificador informal. Això significa que els usuaris podran

canviar el seu nom d'usuari **sempre i quant aquest sigui únic**. L'aplicació s'ha d'encarregar de mantenir la integritat de clau primària per al camp *username*.

Representació única de Missatges

En el model de dades queda molt clar: els missatges tenen una representació única i diversos usuaris assignats. Tanmateix, quan vulguem traslladar aquesta representació cap a el model de dades descentralitzat, no és segur que puguem mantenir aquesta unicitat. És probable que haguem de replicar els missatges per tal d'emmagatzemar-los, i cal decidir quina política prendre respecte als identificadors de missatge en cas que això s'hagi de fer.

Una opció és no replicar els missatges, que aquests mantinguin la seva unicitat i una identificació única. Tanmateix, per a *HorNet* aquesta solució no ens val, donat que necessitem fer una representació descentralitzada i no ens podem comprometre a mantenir un emmagatzematge central. Per tant, optarem per tenir una representació descentralitzada, amb identificadors únics per a cada representació. És a dir que donada una llista de missatges, cada missatge serà únic i identificable dins d'aquella representació, però no podrem saber si dos missatges en diferents representacions son el mateix missatge o no.

Model de dades

El model de dades, després de considerar les funcionalitats, és el següent:

Camps de la classe User

- id: un nombre enter únic que actua d'identificador formal de l'usuari.
- username: un *string*[15] únic que identifica informalment l'usuari.
- password: un *string* únic que serveix de clau d'accés per a l'usuari.
- avatar: un *string* únic que conté la ruta a una URL[16]. Aquesta URL ha de ser accessible des del contenidor, donat que conté la imatge que es veurà en el perfil d'usuari.

Camps de la classe Message

- body: un *string* que conté el cos del missatge.

[15] String: tipus de dada que representa una cadena de caràcters.

[16] URL: sigles de *Uniform Resource Locator*, es el nom genéric que es fa servir per parlar d'adreces, en aquest cas, web.

- date: un *long*[17] que conté la data en el que es va escriure el missatge (calculada com el nombre de mil·lisegons que van passar des de *epoch*[18])

Significat de les relacions

Nom	**publishes**
Classes	User - Messages
Cardinalitat	1 - *
Camps associats	Timestamp: enter que identifica i ordena un missatge per ordre de publicació per un usuari determinat.
Descripció	Relaciona un usuari amb els missatges que ha publicat.

Nom	**isOriginalAuthor**
Classes	User - Messages
Cardinalitat	0..1 - *
Camps associats	-
Descripció	Relaciona un missatge amb el seu autor original. Si no existeix, s'entén que el seu autor original és el que ha publicat el missatge.

Nom	**follows**
Classes	User - User
Cardinalitat	* - *
Camps associats	-
Descripció	Relaciona un usuari amb els usuaris que segueix. La relació no és necessàriament recíproca.

Nom	**mentions**
Classes	User - Messages
Cardinalitat	* - *
Camps associats	Timestamp: enter que identifica i ordena un missatge per ordre de menció a un usuari determinat.
Descripció	Relaciona un usuari amb tots els missatges que el mencionen i viceversa.

[17] *long*: (o long integer) tipus que representa un nombre enter molt gran.

[18] *epoch*: nom que es dona a la data de referéncia agafada per a calcular la data actual. En el cas de Java, *epoch* correspon a l'1 de gener de 1970, a les 12 en punt de la matinada.

Nom	receives
Classes	User - Messages
Cardinalitat	* - *
Camps associats	Timestamp: enter que identifica i ordena un missatge per ordre de recepció en un usuari determinat.
Descripció	Relaciona un usuari amb els missatges que ha rebut i viceversa.

Arquitectura orientada a serveis

Com ja hem comentat en apartats anteriors, la part del servidor està implementada fent servir CoDeS, que implementa un núvol mitjançant una arquitectura orientada a serveis. Per tant, per a definir els requeriments del nostre sistema, haurem de definir una sèrie de serveis que s'encarregaran d'emmagatzemar la informació i facilitar-ne l'accés.

Estructuració de la informació

Cal que escollim com volem estructurar la informació en el sistema abans de decidir quins serveis formaran la nostra arquitectura. Depenent de la manera en com estructurem aquesta informació, els serveis guardaran una cosa o altra, i les operacions de consulta i inserció que tindran seran diferents.

Model completament distribuït

Es tracta del model amb una granularitat més fina. El que volem fer és desplegar diversos serveis per a cada usuari, i que cada un d'aquests serveis s'encarregui de gestionar una part de la informació assignada a aquest usuari. És una solució molt escalable i ampliable, donat que si volem afegir més usuaris només hem de desplegar més serveis, i que si volem afegir noves informacions per a cada usuari ni tan sols hem de modificar els serveis que ja estan en funcionament: n'hi ha prou amb desplegar serveis nous.

Els serveis que despleguem, doncs, estaran associats a dues coses: un usuari i un tipus d'informació. L'identificador del servei, amb el qual es localitzarà dins del sistema, tindrà un format "*USUARI.SERVEI*", on "USUARI" és l'identificador de l'usuari al que està associat, i "SERVEI" una enumeració que ens indica el tipus d'informació que emmagatzema. Així doncs, per a cada usuari desplegarem sis serveis:

- USUARI.Followers: Contindrà la llista d'usuaris que segueixen a USUARI.
- USUARI.Following: Contindrà la llista d'usuaris als que USUARI segueix.

- USUARI.Messages: Contindrà els missatges dels que USUARI és autor.
- USUARI.Timeline: Contindrà el timeline[19] d'USUARI, és a dir, tot el conjunt de missatges que tan ell com els seus seguidors han enviat.
- USUARI.Mentions: Contindrà tots els missatges d'altres usuaris que mencionen a USUARI.
- USUARI.Profile: Contindrà la informació referent al perfil de USUARI.

La granularitat dels serveis, com es pot veure, és la més fina disponible: les dades emmagatzemades en cada un dels serveis no es pot separar de cap manera. Això significa també que si un sol d'aquests serveis està implicat en una petició feta per l'usuari, aquesta no es podrà servir.

Tanmateix, això no és necessàriament un problema. Una de les principals característiques de CoDeS es que permet la replicació de serveis idèntics per a garantir una millor disponibilitat. A més, no cal preocupar-se per que dos serveis idèntics tinguin informació diferent: la gestió de les dades i al comunicació entre dues rèpliques del mateix servei forma part de la gestió pròpia de dades de CoDeS.

Cada un d'aquests serveis, doncs, pot emmagatzemar informació o bé obtenir-la. Exceptuant el servei Profile, on només s'emmagatzema un sol perfil d'usuari, a la resta de serveis s'emmagatzema un conjunt de dades, totes elles de tipus missatge. Per tant, a cada servei hi ha de figurar com a mínim una operació per a emmagatzemar una dada i una operació per a llistar-les totes. Les operacions mínimes de cada servei serien, doncs:

- **Get** *()* : obté tota la informació emmagatzemada en el servei.
- **Insert** *(data)*: emmagatzema una dada al servei.

Model parcialment distribuït

En l'anterior model es despleguen quatre serveis per a cada un dels usuaris, el qual ens proporciona una distribució de la informació francament bona. De tota manera, depenent de quin fos el cost de mantenir o desplegar un servei, ens pot resultar interessant obtenir una distribució menys expansiva. Una opció seria desplegar tota la informació referent a un usuari en un mateix servei.

[19] Timeline: literalment, línia temporal. És el nom pel que es coneix a twitter a la pantalla que recopila tots els missatges que un usuari rep, en ordre temporal d'aparició.

Així, per a cada usuari es desplegaria un servei identificat com a "*USUARI*", on USUARI és l'identificador de l'usuari associat al servei.. Ja no cal especificar el tipus de servei, donat que Les operacions d'aquest servei seran les següents:

- **Get** (param): Obté tota la informació etiquetada com a *param*. Aquest camp pot tenir els següents valors: *Timeline*, *Messages*, *Followers*, *Following*, *Profile* o *Mentions.*
- **Insert** (data, param): Insereix una dada etiquetada com a *param*.

Tal i com es pot veure, el comportament del servei és molt similar al del model anterior. La única diferència es com està distribuïda la informació en un sol node en comptes de diversos. Això ens reporta menys complexitat, per exemple, en les operacions d'alta i baixa d'usuari on cal desplegar o replegar els serveis.

Model centralitzat amb replicació

Un enfocament completament diferent de l'aplicació, seria plantejar-la com si es tractés d'un model centralitzat. Això xoca una mica amb la intenció inicial, que és implementar un model completament descentralitzat, però matisant una mica es pot veure que no es tracta d'una centralització completa.

El que proposa aquest model es desplegar un o diversos serveis anomenats *DATABASE* a la nostra xarxa distribuïda. Aquests serveis actuaran com una capa addicional d'abstracció, mitjançant la qual podem implementar de múltiples maneres l'emmagatzemament de les dades. A efectes de comunicació amb l'aplicació, l'accés a les dades estarà centralitzat (amb replicació per a mantenir la disponibilitat, ja que no tenim servidors dedicats). Tanmateix, a efectes pràctics, si l'emmagatzemament serà centralitzat o distribuït, dependrà de com implementem la capa inferior.

El servei *DATABASE*, doncs, tindrà dues operacions:

- **Get***(user, param):* Obté tota la informació de l'usuari *user* etiquetada com a *param*. Aquest camp pot tenir els següents valors: *Timeline*, *Messages*, *Followers*, *Following*, *Profile* o *Mentions.*
- **Insert** *(data, user, param):* Insereix una dada de l'usuari *user* etiquetada com a *param*. Aquest camp pot tenir els següents valors: *Timeline*, *Messages*, *Followers*, *Following*, *Profile* o *Mentions.*

El model resultant és completament diferent: ja no hem de desplegar serveis tenint en compte la quantitat d'usuaris que hi ha al sistema, sinó que es desplega un nombre constant de serveis, i aquests tenen l'accés a tota la informació de la base de dades. Tanmateix, l'únic que està estrictament centralitzat es l'accés a la informació. El lloc on aquesta s'emmagatzemi dependrà de com implementem el servei, i per tant es pot fer una implementació distribuïda utilitzant aquest model centralitzat.

Model per a HorNet: model completament distribuït

Després de considerar tres tipus de model diferents, hem decidit quedar-nos amb el primer que hem descrit, **el model completament distribuït**.

El model parcialment distribuït no és més que una versió relaxada, i no ens reporta cap canvi significatiu en la complexitat de la implementació. A més, el cost de desplegament d'un servei no és massa elevat, per tant no hi ha molta diferència entre tenir un servei molt gran o un determinat nombre serveis molt petits, si el volum de dades ha de ser el mateix. Només estem dividint el nombre de serveis per una constant, a costa de fer els serveis més pesants per l'ordre d'una constant, i perdre granularitat amb la conseqüent pèrdua del grau de distribució dels recursos.

Per altra banda, el model centralitzat amb replicació, no ens reporta un paradigma diferent del que podem aconseguir amb un sistema centralitzat. Sí que ens permet flexibilitat de cara a una capa inferior, però si al cap i a la fi el que volem és dissenyar un model descentralitzat, l'únic que estem fent es traslladar la complexitat a una altra capa, i afegir més nivells de comunicació intermedis. No creiem que sigui un avenç.

Així doncs, un cop escollit el tipus de model de dades ja podem continuar amb el disseny dels serveis implicats en el model.

Servei User.Messages

Aquest servei s'encarrega d'emmagatzemar tots els missatges que un usuari ha publicat. Cada un dels missatges tindrà uns atributs que caldrà emmagatzemar. Per a tal efecte, emprarem un tipus abstracte de dades anomenat MessageEntry que tindrà els següents camps:

Paràmetre	Tipus de dades	Descripció
Timestamp	Integer	Identificador del missatge dins del servei
UserId	Integer	Identificador de l'usuari que publica el missatge. Es tracta d'un camp redundant: tots els missatges que s'emmagatzemen en aquest servei tenen el mateix usuari.
OriginalUserId	Integer	Identificador de l'autor original del missatge
Body	String	Cos del missatge
Date	Long	Data de publicació del missatge

Les operacions de les que disposa el servei *User.Messages* són les següents:

- **AddMessage** (MessageEntry data). Afegeix el missatge *data* al servei.
- **RemoveMessage** (Integer tstamp). Elimina el missatge identificat amb el timestamp *tstamp* del servei.
- **GetAllMessages** (). Retorna tots els missatges del servei.
- **GetMessageCount** (). Retorna el nombre de missatges publicats al servei.
- **GetXNewerItems** (Integer count). Retorna els *count* últims missatges publicats al servei.
- **GetItemsNewerThanTS** (Integer tstamp). Retorna els missatges amb un timestamp més gran que *tstamp*.
- **GetItemsNewerThanDate** (Long date). Retorna els missatges publicats després de la data *date*.

Servei User.Timeline

Aquest servei emmagatzema tots els missatges que un usuari ha rebut. És a dir, un conjunt que inclou: els missatges que el propi usuari ha publicat, els missatges que els *following* de l'usuari han publicat, i els missatges en els que l'usuari és mencionat. Igual que en el cas anterior del servei Messages, cal definir un tipus abstracte de dades per a emmagatzemar les entrades del contingut del servei. Anomenarem a aquest tipus de dades TimelineEntry.

Paràmetre	Tipus de dades	Descripció
Timestamp	Integer	Identificador del missatge dins del servei
OriginalTstamp	Integer	Identificador del missatge dins del servei messages de l'autor.
UserId	Integer	Identificador de l'usuari que publica el missatge
OriginalUserId	Integer	Identificador de l'autor original del missatge
Body	String	Cos del missatge
Date	Long	Data de publicació del missatge

Les operacions de les que disposa el servei *User.Timeline* són les següents:

- **AddMessage** (TimelineEntry data). Afegeix el missatge *data* al servei.
- **RemoveMessage** (Integer tstamp). Elimina el missatge identificat amb el timestamp *tstamp* del servei.
- **GetAllMessages** (). Retorna tots els missatges del servei.
- **GetXNewerItems** (Integer count). Retorna els *count* últims missatges publicats al servei.
- **GetFromXToY** (Integer x, Integer y). Retorna els missatges amb timestamps compresos entre *x* i *y*.
- **GetItemsNewerThanTS** (Integer tstamp). Retorna els missatges amb un timestamp més gran que *tstamp*.
- **GetItemsNewerThanDate** (Long date). Retorna els missatges publicats després de la data *date*

Tot i que s'assembla molt al servei *User.Messages*, cal tenir molt presents les diferències. En el servei *User.Timeline* no hi ha una funció que ens retorna el nombre de items, per un motiu molt simple: per raons d'espai, al servei *User.Timeline* no s'emmagatzemen els missatges més antics, tan sols un nombre finit dels missatges més recents. Això és degut, per una banda, a que la immensa majoria de gent no es para a llegir tots els missatges que ha rebut si aquest nombre és massa elevat. I per altra banda, cal tenir en compte que el ritme de creixement del volum de dades dels serveis *User.Timeline* és exponencial respecte al ritme de creixement del volum de dades dels serveis *User.Message*. Emmagatzemar tots els missatges faria que la mida del servei acabés sent insostenible, en comunitats amb molts usuaris.

L'altre canvi respecte al servei *User.Messages* és la operació *GetFromXToY*. Fins i tot limitant el nombre de missatges que s'emmagatzemen, se'n desen prous com perquè treballar sempre amb la totalitat de missatges no sigui recomanable. És per això que aquest servei està preparat per treballar amb blocs de missatges, i així poder-los carregar poc a poc i sota demanda.

Servei User.Followers

Aquest servei s'encarrega d'emmagatzemar els usuaris que estan seguint a l'usuari associat al servei. Al contrari dels dos serveis que ja hem vist, en aquest servei

no s'emmagatzemen estructures complexes de dades. Les entrades que aquí s'emmagatzemen són simples enters, corresponents a l'identificador de l'usuari al que fa referència la relació de *follow*.

Les operacions que disposem per a aquest servei són:

- **AddUser** (Integer user). Afegeix l'usuari *user* al servei.
- **RemoveUser** (Integer user). Elimina l'usuari *user* del servei.
- **ContainsUser** (Integer user). Retorna "cert" en cas que l'usuari *user* consti en el servei, i "fals" altrament.
- **GetUserCount** (). Torna el nombre total d'usuaris que figuren en el servei.
- **GetAllUsers** (). Torna la llista completa d'usuaris que figuren en el servei.
- **GetXNewerUsers** (Integer count). Torna els *count* últims usuaris que s'han afegit al servei.

Servei User.Following

De manera anàloga al servei *User.Followers*, aquest servei s'encarrega d'emmagatzemar la referéncia d'usuaris als que l'usuari associat al servei està subscrit. Les operacions amb les que treballa són exactament les mateixes. Així doncs, tenim aquesta llista:

- **AddUser** (Integer user). Afegeix l'usuari *user* al servei.
- **RemoveUser** (Integer user). Elimina l'usuari *user* del servei.
- **ContainsUser** (Integer user). Retorna "cert" en cas que l'usuari *user* consti en el servei, i "fals" altrament.
- **GetUserCount** (). Torna el nombre total d'usuaris que figuren en el servei.
- **GetAllUsers** (). Torna la llista completa d'usuaris que figuren en el servei.
- **GetXNewerUsers** (Integer count). Torna els *count* últims usuaris que s'han afegit al servei.

Servei User.Mentions

Aquest servei emmagatzema tots els missatges en els que l'usuari associat ha sigut mencionat. Al emmagatzemar missatges de diferents autors, el servei es pràcticament idèntic al servei *User.Timeline*. Per a emmagatzemar les entrades, fem servir un tipus abstracte de dades anomenat MentionEntry que emmagatzema aquesta informació:

Paràmetre	Tipus de dades	Descripció
Timestamp	Integer	Identificador del missatge dins del servei
OriginalTstamp	Integer	Identificador del missatge dins del servei messages de l'autor.
UserId	Integer	Identificador de l'usuari que publica el missatge
OriginalUserId	Integer	Identificador de l'autor original del missatge
Body	String	Cos del missatge
Date	Long	Data de publicació del missatge

Les operacions de les que disposa el servei *User.Mentions* són les següents:

- **AddMessage** (TimelineEntry data). Afegeix el missatge *data* al servei.
- **RemoveMessage** (Integer tstamp). Elimina el missatge identificat amb el timestamp *tstamp* del servei.
- **GetAllMessages** (). Retorna tots els missatges del servei.
- **GetXNewerItems** (Integer count). Retorna els *count* últims missatges publicats al servei.
- **GetFromXToY** (Integer x, Integer y). Retorna els missatges amb timestamps compresos entre *x* i *y*.
- **GetItemsNewerThanTS** (Integer tstamp). Retorna els missatges amb un timestamp més gran que *tstamp*.
- **GetItemsNewerThanDate** (Long date). Retorna els missatges publicats després de la data *date*

Servei User.Profile

Aquest servei s'encarrega d'emmagatzemar la informació referent a l'usuari. A diferència dels serveis que hem vist fins ara, aquest no s'encarrega d'emmagatzemar llistes d'informació, sinó que emmagatzema el perfil d'un únic usuari. Representarem aquest perfil mitjançant un tipus abstracte de dades que anomenarem *ProfileEntry*.

Paràmetre	Tipus de dades	Descripció
Id	Integer	Identificador numèric de l'usuari
Username	String	Nom de l'usuari, identificador informal
Password	String	Contrasenya de l'usuari
Avatar	String	URL contenint l'avatar de l'usuari
Published	Boolean	Val "cert" si l'usuari accedeix a fer el seu perfil públic, "fals" altrament.
Keys	List<String>	Llista els noms dels paràmetres opcionals del perfil
Values	Map	Permet emmagatzemar els valors dels paràmetres opcionals del perfil en parelles <nom, valor>.
IndividualPub	List<Boolean>	Indica, per a cada paràmetre opcional, si l'usuari accedeix a fer-lo públic.

Les operacions del servei *User.Profile* són molt senzilles:

- **PutProfile** (ProfileEntry data). Carrega el perfil *data* al servei.
- **GetProfile** (). Retorna el perfil de l'usuari associat al servei.

Consideracions sobre els serveis

Els serveis descrits en l'apartat anterior a priori són suficients per a fer funcionar el programa d'una manera segura i eficaç. Tanmateix, hi ha un parell de casos en les funcionalitats que hem definit en els quals descobrim que potser no funcionaria d'una manera massa eficient, fins al punt que operacions que en un entorn centralitzat serien molt senzilles i ràpides, aquí es tornarien molt lentes i costoses.

El problema dels *logins*

Al fer un simple *login*, l'únic que caldria és anar al servei *User.Profile* associat a l'usuari per al que volem fer login, i comprovar que la seva contrasenya coincideix. Tanmateix, ens trobem en que l'usuari introdueix en el sistema el seu identificador informal (username), i nosaltres hem identificat els serveis mitjançant un identificador formal (id) per donar més versatilitat als canvis de nom d'usuari. La única manera que tenim de conèixer l'identificador numèric a partir del nom d'usuari és consultar perfil per perfil fins trobar una coincidència. No cal dir que en un entorn amb molts usuaris aquesta operació pot ser extremadament lenta.

El problema de les cerques

Quan intentem fer una cerca d'usuari ens trobem exactament el mateix problema que acabem de veure: hauriem d'anar consultant perfil per perfil,

comprovant els valors dels camps que han sigut publicats anant enumerant les coincidències que anem trobant per a retornar la llista definitiva.

Solució: serveis addicionals

Per tal d'evitar els problemes de logins i cerques, cal que despleguem un parell de serveis addicionals: un servei *Authentication* que s'encarregui de mantenir una relació username-id, i un servei *Indexation* que s'encarregui de centralitzar la informació que els usuaris publiquen al seu perfil, per tal que les cerques siguin més ràpides. La cardinalitat d'aquests serveis serà constant: no tindran un usuari concret associat, per tant no dependrà del nombre d'usuaris que hi hagi al sistema.

Servei Authentication

Aquest servei s'encarrega de mantenir una relació clau-valor per tal que es pugui obtenir de manera més ràpida l'identificador d'usuari a partir del seu nom, que és una operació que es realitzarà moltes vegades tenint en compte que l'aplicació, des del punt de vista de la interfície d'usuari, identifica els usuaris per l'*username* i no per l'*id*.

A més, també hem afegit altres operacions per tal de facilitar, per exemple, la operació d'autenticació que s'encarrega de comprovar que una contrasenya sigui correcta per a un nom d'usuari concret, o també una operació per a controlar quin identificador d'usuari donar a un usuari nou per tal de garantir-ne la unicitat.

De la mateixa manera que hem considerat en els serveis que s'encarreguen de gestionar altres estructures de dades, per a aquest servei utilitzarem un tipus abstracte de dades anomenat *AuthEntry* que tingui els següents camps:

Paràmetre	Tipus de dades	Descripció
Id	Integer	Identificador numèric de l'usuari
Username	String	Nom de l'usuari, identificador informal
Password	String	Contrasenya de l'usuari

Les operacions del servei són les següents:

- **InsertUser** (AuthEntry user). Afegeix l'usuari *user* al servei. Si l'identificador existeix en el servei, en modifica el nom d'usuari i el password perquè s'ajustin als valors de *user*.
- **RemoveUser** (Integer id). Elimina l'usuari amb identificador *id* del servei.

- **GetAllUsers** (). Obté tots els usuaris emmagatzemats en el servei.
- **GetUserAuth** (String username, String password). Si la contrasenya de *username* es *password*, retorna l'identificador de l'usuari, o un error altrament.
- **GetUserId** (String username). Retorna l'identificador numéric associat al nom d'usuari *username*.
- **GetNextId** (). Retorna un identificador no existent en el servei.

Servei Indexation

Aquest servei s'encarregarà de relacionar un identificador d'usuari amb un conjunt d'informació, decidit pel propi usuari, que el definirà de cara a una cerca. Treballarem, doncs, amb una estructura de dades representada pel tipus abstracte de dades *IndexEntry* amb els següents camps:

Paràmetre	Tipus de dades	Descripció
Id	Integer	Identificador numèric de l'usuari
Information	List<String>	Informació de cerca relacionada amb l'usuari

Les operacions del servei son les següents:

- **Publish** (IndexEntry information). Crea una nova entrada *information* en el servei, o bé si l'identificador d'usuari associat al paràmetre d'entrada ja existia, n'actualitza la informació de cerca relacionada.
- **RemoveUser** (Integer id). Elimina l'entrada associada a l'identificador *id* del servei.
- **Search** (String pattern). Retorna la llista d'usuaris que contenen la cadena *pattern* dins de la seva informació de cerca relacionada.

Especificació del client

De manera similar a Twitter, volem que HorNet disposi d'un client web que permeti a l'usuari comunicar-se amb l'aplicació mitjançant una interfície d'usuari amigable. Per a implementar aquest client, hem escollit utilitzar la tecnologia de *Java Servlets*, per una banda per la facilitat que ens suposa de cara a la comunicació amb CoDeS (donat que aquest també està implementat en Java), i per altra banda per la modularitat i versatilitat que ens permet aquesta tecnologia.

En aquesta secció del document descriurem com hem estructurat aquest client web, quins components té i com es realitza la comunicació amb ells.

Introducció: Servlets en Jetty

Com ja hem comentat en l'apartat de les tecnologies usades, hem decidit utilitzar *Jetty* com a contenidor per a allotjar el nostre client. Aquest client, a més de ser de codi obert, compta amb un avantatge addicional: existeix una versió de Jetty preparada expressament per a funcionar integrat en CoDeS, el qual ens pot permetre una millor integració a la xarxa.

Volem fer una petita introducció a com es tracten aquests components per tal d'entendre millor com funcionarà el nostre client web.

Servlets

Hem vist que l'aplicació està implementada utilitzant *Servlets*. Tanmateix, què és un Servlet?

Un Servlet és una petita aplicació de java que hereta de la classe *HttpServlet*. Aquesta classe es dedica a rebre peticions HTTP, tractar-les i retornar una resposta. Per a tal efecte, té dos mètodes diferents que cal sobrecarregar:

- **doGet**: aquesta funció es cridarà quan al servlet arribi una petició HTTP amb un mètode *GET* a la seva capçalera.
- **doPost**: de manera anàloga, aquesta funció es cridarà quan el mètode de la petició sigui *POST*.

Tan un mètode com l'altre tenen dos paràmetres: un de tipus *HttpServletRequest* i un de tipus *HttpServletResponse*. Del paràmetre *request* s'agafa

la informació d'entrada, com ara els paràmetres de la petició HTTP o la sessió, i del paràmetre *response* s'agafen les estructures necessàries per a retornar la resposta.

La manera d'utilitzar aquests servlets, doncs, és molt senzilla. Cal mapejar-los en una localització per tal de fer-los accessibles per al navegador, i llavors se'ls pot enviar peticions per a que aquests les interpretin. Aquest mapeig, en el cas de Jetty, es realitza mitjançant una classe anomenada *Server*, i en el cas de *HorNet* ho hem implementat en una classe que hem anomenat *JettyLauncher*.

La classe JettyLauncher

La classe principal del client de HorNet és una classe executable[20] que s'encarrega principalment de proporcionar a Jetty les dades necessàries per a que sàpiga com executar HorNet.

El codi d'aquesta classe realitza els següents passos:

1. Crea una nova instància de la classe *org.mortbay.jetty.Server*, afegint-li un listener[21] al port 8070. Això significa que ens connectarem al client de HorNet a través del port 8070, en comptes de pel port 80 com és habitual.
2. Obté el context "/" del servidor que acabem de crear. Aquest context representa el directori arrel de l'adreça a la que ens connectem.
3. Mapeja, en aquest context, els servlets necessaris per a HorNet.
4. Finalment, inicialitza el servidor web. L'execució de JettyLauncher queda bloquejada, i interrompre-la significaria perdre connexió amb el servidor web.

El mapeig de servlets necessari per al correcte funcionament de HorNet és el següent:

[20] Classe executable: Classe de java que conté un mètode main. El resultat és que es pot executar des de la línea de comandes.

[21] listener: component que "escolta" en una determinada via d'informació fins a l'arribada d'un event.

Servlet	Mapeig
LoginServlet	/
RegisterServlet	/register
MainServlet	/main
SendTweetServlet	/sendTweet
ReceivedMessagesServlet	/update
SearchServlet	/search
SelfProfileServlet	/selfprofile
ViewProfileServlet	/profile
FollowServlet	/follow

Això significa que, per exemple, si volem accedir a *RegisterServlet*, l'únic que cal que fem és escriure la ruta */register* després del host en el nostre navegador. D'aquesta manera, el Servlet captarà la capçalera HTTP, farà les operacions necessàries, i generarà una resposta.

Enllaços vs Redireccions vs Formularis

Abans de començar a veure les especificacions dels *Servlets*, cal tenir en compte una petita consideració. Des del punt de vista d'una aplicació web, podem accedir a un servlet de tres maneres diferents. Cada una d'elles té uns avantatges i uns inconvenients, que cal tenir en compte de cara a saber com i quan utilitzar-les:

- **Enllaços.** Un enllaç o *hyperlink* consisteix en un fragment de text que, al clicar, envia al servidor una petició HTTP. És la manera més freqüent de bellugar-se per una pàgina web. Aquesta acció és interactiva, és a dir, requereix una acció per part de l'usuari, i només serveix per enviar peticions HTTP amb el mètode *GET*.
- **Redireccions.** Tenen la mateixa base que els hipervincles, envien una petició HTTP però la diferència és que no és l'usuari el que l'activa sinó un *script*[22] intern de la pròpia aplicació i per tant no requereixen d'interacció per part de l'usuari. Tanmateix, exactament igual que en el cas dels enllaços, només permeten el mètode *GET*.
- **Formularis.** Un formulari ens permet generar una petició HTTP molt més rica, ja que no només ens permet escollir entre el mètode *GET* i el mètode *POST* segons ho necessitem, sinó que a més ens permeten enviar informació interactivament a través de camps d'entrada o *input* associats al propi formulari.

[22] script: fragment de codi simple escrit en un llenguatge de programació, normalment, interpretat.

Especificació dels Servlets de HorNet

Ja hem pogut veure una llista dels servlets que són necessaris per a *HorNet* i on es troben mapejats per al seu accés. En aquesta secció volem especificar els detalls sobre cada un dels servlets: quina entrada s'esperen, què fan i quina resposta tornen.

LoginServlet

Aquest Servlet és el primer que l'usuari veu quan es connecta al host que allotja *HorNet*, donat que es troba mapejat en la ruta "/". S'encarrega principalment de mostrar la pantalla de *login*, processar les peticions d'autenticació i redirigir cap a la pantalla principal.

Mètode get

Aquest mètode ignora completament els paràmetres d'entrada de la petició HTTP. Simplement mostra un enllaç cap a la ruta "/register" (*RegisterServlet*) i un formulari amb dos camps (nom d'usuari i contrasenya) i un botó per a que l'usuari pugui autenticar-se. Aquest formulari s'envia a la ruta "/" (*LoginServlet*) mitjançant el mètode *POST*.

Mètode post

Aquest mètode agafa dos paràmetres d'entrada: **uname** i **pass**, i intenta autenticar-se a HorNet amb ells. Si ho aconsegueix, retorna una pantalla que ens redirigeix cap a la ruta "/main" (*MainServlet*). Si no, ens redirigeix de nou cap a la ruta "/" (*LoginServlet*) per a que ho tornem a intentar.

RegisterServlet

S'accedeix a aquest Servlet mitjançant un enllaç ubicat a la ruta "/" (*LoginServlet*) i és el que utilitzen els usuaris que acaben d'entrar a la comunitat per a crear-se un nou compte d'usuari.

Mètode get

Aquest mètode ignora completament els paràmetres d'entrada de la petició HTTP. Simplement mostra un enllaç per a tornar cap a la ruta "/" *(LoginServlet)* i un formulari amb tres camps (nom d'usuari, contrasenya i avatar) i un botó per a que l'usuari pugui autenticar-se. Aquest formulari s'envia a la ruta "/register" (*RegisterServlet*) mitjançant el mètode *POST*.

Mètode post

Aquest mètode agafa tres paràmetres d'entrada: **uname**, **pass** i **avatar** i intenta crear un nou usuari de HorNet amb ells. Si ho aconsegueix, retorna una pantalla amb un missatge d'èxit i una redirecció cap a la ruta "/" (*LoginServlet*) per a que puguem entrar al sistema, i si no ens retorna una pantalla amb un missatge d'error i un enllaç cap a la ruta "/register" (*RegisterServlet)*.

MainServlet

Aquest Servlet és el que conté la immensa majoria dels continguts de l'aplicació. No reacciona al mètode POST, i el mètode GET ignora els paràmetres d'entrada de la request HTTP. En aquest servlet es poden trobar els medis per a realitzar les següents tasques:

- **Consultar el propi perfil.** Mostra un enllaç cap a la ruta "/profile?u={name}" (*ProfileServlet)*.
- **Editar el propi perfil**. Mostra un enllaç cap a la ruta "/selfprofile" (*SelfProfileServlet*).
- **Desconnectar-se.** Mostra un enllaç cap a la ruta "/" (*LoginServlet*). La sessió es neteja en el procés, provocant una desconnexió de l'usuari actual.
- **Enviar un missatge**. Mostra un camp de text i un botó que envia el formulari cap a la ruta "/sendTweet" (*SendTweetServlet*). La resposta a aquesta petició es mostra en una etiqueta situada a sota del formulari.
- **Consultar el timeline.** Mitjançant una crida en un script, permet refrescar periòdicament els missatges que un usuari ha rebut mitjançant crides a la ruta "/update" (*ReceivedMessagesServlet*).
- **Fer cerques**. Mostra un camp de text i un botó que envia el formulari cap a la ruta "/search" (*SearchServlet*) a través del mètode *POST*.
- **Consultar els *followers* i *following*.** Mitjançant una crida en un script, permet refrescar periòdicament els usuaris que segueixen o als que segueix l'usuari mitjançant crides a la ruta "/follow?op=list" (*FollowServlet*) a través del mètode *GET*.

SendTweetServlet

Aquest servlet no reacciona al mètode GET, tan sols té implementada la funció *doPost*. De la petició HTTP agafa un paràmetre d'entrada, **mess**, i amb això i la informació de sessió de l'usuari que està utilitzant el client, intenta crear un missatge

nou. En cas de tenir èxit, inclou a la resposta el missatge "*Message sent successfully*", i altrament retorna un missatge d'error.

ReceivedMessagesServlet

Aquest servlet no reacciona al mètode POST, tan sols té implementada la funció *doGet*. De la petició HTTP només agafa informació referent a l'usuari que està connectat, i amb aquesta informació fa una petició a CoDeS per a obtenir el *timeline* d'aquell usuari. Un cop ha obtingut la informació, li dona format HTML mitjançant la funció privada *message* i la inclou a la resposta per a que aquesta es pugui mostrar per pantalla.

SearchServlet

Aquest servlet no reacciona al mètode GET, tan sols té implementada la funció *doGet*. De la petició HTTP només agafa el paràmetre **q**, i amb aquest paràmetre fa la petició a CoDeS per a realitzar la cerca. Un cop té la informació, li dona format HTML mitjançant la funció privada *user* i la inclou a la resposta per a que aquesta es pugui mostrar per pantalla.

SelfProfileServlet

Aquest servlet s'encarrega de gestionar l'edició del propi perfil d'un usuari. S'hi accedeix des de la ruta "/selfprofile", i com la majoria de servlets que implementen el mètode *doGet*, està preparat per a mostrar informació orientada a usuari.

Mètode Get

Aquest mètode mostra un formulari amb molts camps: un per a cada un dels atributs de la classe User. Aquest formulari s'envia, mitjançant un botó, a la ruta "/selfprofile" (*SelfProfileServlet*). També mostra un enllaç a la ruta "/main" (*MainServlet*) utilitzada per a tornar enrere.

Mètode Post

Aquest mètode agafa un conjunt de paràmetres de la petició HTTP i intenta construir un objecte de la classe *User* amb ells. Les possibles respostes que pot retornar son "Passwords don't match" si les contrasenyes no coincideixen, "Incorrect Password" si la contrasenya actual no és correcta, "Username is not valid" si es produeix una infracció de clau primària amb el nou nom d'usuari, o bé "Profile updated successfully" si tot va correctament. Un cop l'objecte està creat, el fa arribar a CoDeS

mitjançant una crida. Si l'usuari ha decidit publicar informació, també fa una altra crida a CoDeS per a comunicar quina és aquesta informació.

ViewProfileServlet

Aquest servlet no reacciona davant peticions fetes al mètode POST. Pel que fa al mètode GET, s'encarrega de mostrar una pàgina amb informació sobre el perfil d'un usuari en concret, obtingut a partir del valor del paràmetre **u** a la petició HTTP. Aquesta informació consta, principalment, de:

- La informació del perfil que l'usuari ha decidit fer pública.
- Els missatges que aquell usuari ha escrit
- Un botó per a fer-se o deixar de ser seguidor d'aquell usuari. El botó envia una petició GET a la ruta "/follow" (*FollowServlet)*.

FollowServlet

Aquest servlet no respon davant peticions fetes al mètode GET. Pel que fa al mètode POST, obté el valor del paràmetre **op** de la capçalera HTTP i en comprova el valor. Depenent de quin sigui aquest valor, es comportarà d'una manera o una altra.

- **op = list**. En aquest cas, el servlet mostra la llista completa de *followers* i *followings* corresponents a l'usuari que ha iniciat sessió, degudament formatada en HTML.
- **op** = **f**. (**f**ollow). En aquest cas, el servlet comprova el valor del paràmetre **t** (**t**arget) de la petició HTTP, i fa una crida a *CoDeS* per a establir una relació de seguiment.
- **op = uf**. (**u**n**f**ollow). En aquest cas, el servlet comprova el valor del paràmetre **t** (**t**arget) de la petició HTTP, i fa una crida a CoDeS per a trencar una relació de seguiment.

Diagrama de navegabilitat

El diagrama de navegabilitat que conformen els *Servlets*, expressat d'una manera gràfica, té aquest aspecte:

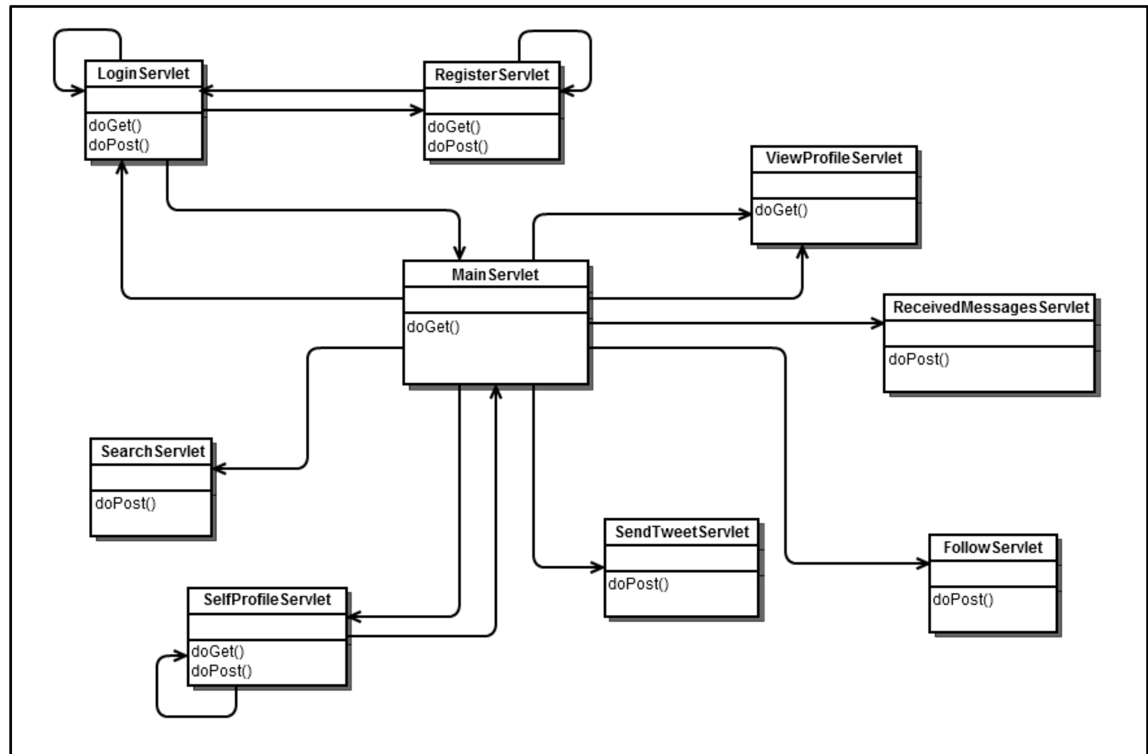


Diagrama de navegabilitat (font: pròpia)

La connexió client-servidor

Hem pogut veure amb detall les especificacions dels Servlets, i els comportaments que tenen: quines sortides retornen segons l'entrada que se'ls subministri. Tanmateix, hi ha un pas molt important que encara no hem vist, i és com es realitza la comunicació entre els Servlets, i el servidor, en la seva implementació en forma de serveis de CoDeS.

Per a realitzar aquesta comunicació, hem optat per a implementar una classe intermèdia que actuï de mèdium. D'aquesta manera contribuïm a una millor modularitat del programa, donat que els *servlets* no necessiten conèixer la implementació del servidor, i es poden permetre centrar-se exclusivament en el tractament d'entrades i generació de sortides.

En aquesta secció veurem els detalls respecte a aquesta classe intermèdia, la seva especificació, i els detalls de la implementació que hem escollit per a CoDeS.

La interfície DataManager

Per tal de facilitar la integració de diverses implementacions diferents del nexe client-servidor, hem decidit definir una interfície per a assegurar que les implementacions que derivin tinguin els mètodes adequats.

Els mètodes que es defineixen en aquesta interfície són els següents:

- **getUserAuth** (String username, String password). Retorna un objecte de la classe *User* corresponent a les credencials *username* i *password*, o bé un valor nul si l'autenticació ha fallat.
- **registerUser** (String username, String pass, String avatar). Genera un usuari amb les credencials indicades per *username*, *pass* i *avatar*, i el registra al sistema. Aquesta funció retorna un valor enter, corresponent a l'identificador numèric de l'usuari recent creat.
- **updateProfile** (User user). Actualitza el perfil d'un usuari amb el contingut del paràmetre d'entrada *user*. Aquesta funció no retorna res.
- **insertAuth** (int id, String user, String pass). Inserta (o actualitza si ja existia) el valor d'autenticació de l'usuari identificat amb *id*, amb els valors *user* i *pass*. Retorna un booleà que equival a cert si la operació s'ha produït sense problemes o bé fals si s'ha produit algun error.
- **postTweet** (int id, String message). Inserta un missatge, amb identificador d'usuari *id*, cos de missatge *message*, usuari original amb identificador "0" i data corresponent al dia i hora actuals. Aquesta funció efectua totes les relacions necessàries perquè el missatge sigui accessible per a tothom que hi tingui dret, i retorna un enter, corresponent al *timestamp* del missatge recent creat.
- **retweet** (int id, TlEntry message). Inserta una cópia del missatge *message* al sistema, on l'usuari original equival a l'autor de *message*, i l'autor del missatge equival a *id*. Retorna un enter, corresponent al timestamp del missatge recent creat.
- **getReceived** (int user). Retorna un objecte de tipus java.util.Vector, contenint els missatges del *Timeline* de l'usuari amb identificador *user*.
- **search** (String pattern). Retorna un objecte de tipus java.util.Vector, contenint tots els usuaris (emmagatzemats com a objectes de classe *User*) que contenen la cadena *pattern* dins la seva informació publicada per a cerca.

- **getUserByName** (String name). Retorna un objecte de tipus *User*, corresponent a l'identificador informal (nom d'usuari) *name*.
- **getUserById** (int id). Retorna un objecte de tipus *User*, corresponent a l'identificador formal (identificador d'usuari) *id*.
- **follows** (int id1, int id2). Retorna cert si l'usuari amb identificador *id1* és seguidor de l'usuari amb identificador *id2*, o fals altrament.
- **getFollowing** (int id). Retorna un objecte de tipus java.util.Vector, contenint els usuaris (objectes de tipus User) als que està subscrit l'usuari amb identificador *id*.
- **getFollowers** (int id). Retorna un objecte de tipus java.util.Vector, contenint els usuaris (objectes de tipus User) que estan subscrits a l'usuari amb identificador *id*.
- **doFollow** (int id1, int id2). Afegeix al servidor la relació "*id1* segueix a *id2*".
- **doUnfollow** (int id1, int id2). Elimina del servidor la relació "*id1* segueix a *id2*".

La classe CoDeSDataManager

La implementació que hem escollit per a la interfície *DataManager* a *HorNet* es troba a la classe *CoDeSDataManager*. Aquesta classe és un *singleton*[23] que només es dedica a implementar els mètodes heretats de la interfície, per tal que aquests es sàpiguen comunicar amb . Tanmateix, com a bon *singleton*, té un parell de mètodes que necessita implementar per a garantir que només existeix una instància de la classe.

- **init** (). S'encarrega d'inicialitzar la instància de la classe CoDeSDataManager.
- **getInstance** (). Retorna un objecte de tipus *CoDeSDataManager*, corresponent a la única instància de la classe.

Cal remarcar que, per a la comunicació amb CoDeS, aquesta classe és una subclasse de *ServiceInvokingApp*, una classe que s'explicarà amb més detall en la memòria, en l'apartat de comunicació de l'especificació del servidor.

Altres components del client

Ja hem vist els elements principals que integren el client web, i la seva especificació. Tanmateix, hi ha altres components secundaris que també ajuden a que

[23] singleton: classe de la qual només pot existir una sola instància

la correcta execució del client. Aquests components poden ser classes de java addicionals, o arxius de text que proporcionen paràmetres a les classes del client.

Classe Cache

Una de les operacions que més es repeteix en una xarxa social d'aquestes característiques és la obtenció del perfil d'un usuari. Hom pot pensar que aquesta operació només es produeix quan algú demana activament visitar el perfil d'un usuari, però això no és cert. Al mostrar-se els missatges del *Timeline*, per exemple, cal obtenir els perfils de cada un dels usuaris que figuren en el timeline per tal de visualitzar correctament el seu nom d'usuari i el seu avatar. Cal tenir en compte que aquesta operació, merament visual, té un elevat cost en una arquitectura orientada a serveis com HorNet, donat que cada petició a perfil es realitza mitjançant una operació de xarxa. L'actualització del *Timeline* és força comú, i els usuaris que s'hi veuen reflexats estan repetits moltes vegades, amb el qual es realitzen les mateixes operacions una i altra vegada, provocant un possible col·lapse de xarxa.

Donat que no ens interessa generar una gran quantitat de tràfic per a una operació merament estètica, hem optat per afegir la implementació d'una classe intermèdia anomenada *Cache* que s'encarregui de gestionar un cert nombre d'usuaris coneguts en memòria de contenidor Jetty.

Quan l'aplicació desitja obtenir una instància de la classe *User* a partir d'un identificador d'usuari, no fa la petició directament a la classe *CoDeSDataManager* sinó que ho fa a la classe *Cache*. Aquesta comprova si té l'usuari en memòria i, en cas que no el tingui, és la classe *Cache* la que s'encarrega de fer la petició a *CoDeSDataManager*. Un cop l'obté, l'emmagatzema en memòria i el retorna a l'aplicació que l'havia sol·licitat.

La incorporació d'aquesta classe ens reporta uns avantatges substancials per a obtenir perfils d'usuari que es repeteixen sovint, però té una limitació important: les dades que s'emmagatzemen a la cache no tenen per què estar actualitzades al 100%, donat que no té cap manera de detectar si un usuari ha modificat el seu perfil i aquest no ha sigut recarregat a Cache. Per a aspectes merament visuals com la obtenció de perfils d'usuaris per a mostrar missatges, aquesta limitació no és important. Tanmateix, per a altres usos com per a consultar el perfil activament, potser ens convé molt més efectuar directament la petició a *CoDeSDataManager* sense passar per *Cache*.

Arxiu style.css

El client web de HorNet té un estil ric, aconseguit mitjançant *CSS*. Per a la majoria de *servlets* que disposen d'un mètode Get, el *servlet* carrega el contingut d'un arxiu, anomenat **style.css**, que s'encarrega de donar format als diferents components html que componen la pàgina web. El contingut d'aquest arxiu es diposita a la capçalera HTML, entre els tags <*style*> i </*style*>.

Arxiu ProfileTemplate.xml

Quan parlàvem del disseny dels perfils d'usuari en el capítol anterior, dèiem que no tots els camps del perfil estaven definits, només uns de concrets que eren estrictament necessaris per a la correcta execució de les operacions de HorNet. La resta de camps eren opcionals, i servien per a generar perfils més rics amb possibilitats de cerques més avançades, i era el propi client el qui els definia. La definició d'aquests camps es troba en el fitxer *ProfileTemplate.xml*, i alguns *servlets* del client web el llegeixen per a consultar els valors que s'han configurat per a la comunitat.

L'arxiu *ProfileTemplate.xml* té un aspecte similar a aquest:

```
  <?xml version="1.0" ?>
- <profile_fields>
  - <field key="Real_name" publishable="true">
      <input type="text" />
      <data type="text" />
    </field>
  - <field key="Gender" publishable="false">
    - <input type="select">
        <option value="Male" />
        <option value="Female" />
      </input>
      <data type="text" />
    </field>
  - <field key="Year_of_birth" publishable="true">
      <input type="text" />
      <data type="integer" minlim="1800" />
    </field>
  </profile_fields>
```

ProfileTemplate.xml (font: pròpia)

A part de l'etiqueta <*profile_fields*> que engloba el cos de l'arixu XML, l'estructura d'aquest arxiu és la següent:

- **<field>** Té els atributs *key* i *publishable*. *Key* ens indica amb quin nom de clau s'emmagatzemarà el contingut d'aquest camp en el perfil d'usuari, i *publishable* ens indica si l'usuari té possibilitats de fer aquest camp públic per a la cerca.
- **<input>** Té l'atribut *type*, que pot ser de dos tipus: "text" o bé "select". Aquesta etiqueta marca quina forma tindrà el camp en el formulari del perfil: si serà un camp de text ("text") o si serà un desplegable amb opcions ("select").
- **<option>** (només si *input* és de tipus *select*) Té l'atribut *value*, que ens especifica el valor de cada una de les cadenes entre les que volem optar.
- **<data>** Té l'atribut *type*, que pot ser de tipus "text" o "integer" i ens marca el tipus de dada que s'emmagatzemarà en el camp. En el cas del tipus "integer", es disposa també dels atributs *minlim* i *maxlim*, que validen un valor del camp segons si es troba dins les cotes subministrades.

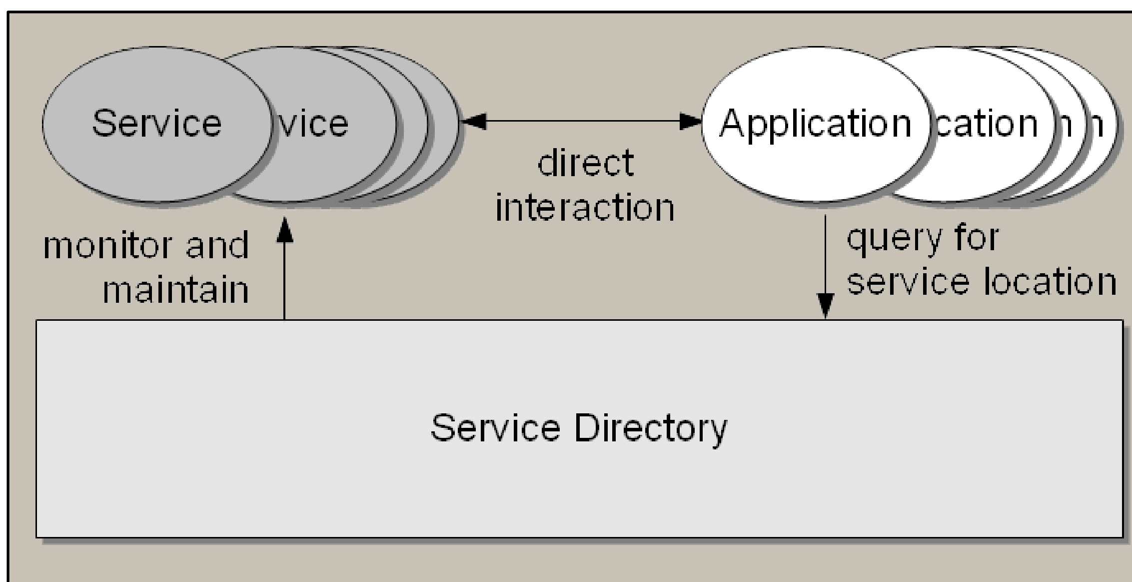
Especificació del servidor

Com es comentava al principi del document, en la secció on s'explicaven les tecnologies a utilitzar, per a implementar l'arquitectura orientada a serveis necessària per a la gestió i emmagatzemament de dades hem escollit CoDeS, un projecte de codi lliure desenvolupat per estudiants universitaris de Catalunya. Per tal d'utilitzar-la, hem desenvolupat una sèrie de serveis compatibles amb CoDeS i hem implementat una interfície per a la nostra aplicació que sigui capaç de comunicar-se amb aquests serveis.

En aquesta secció del document es descriuen tots els detalls referents al component servidor de *HorNet*, tan en el software *third-party* [24]que hem utilitzat, com els que hem implementat des de zero per a tal efecte.

CoDeS

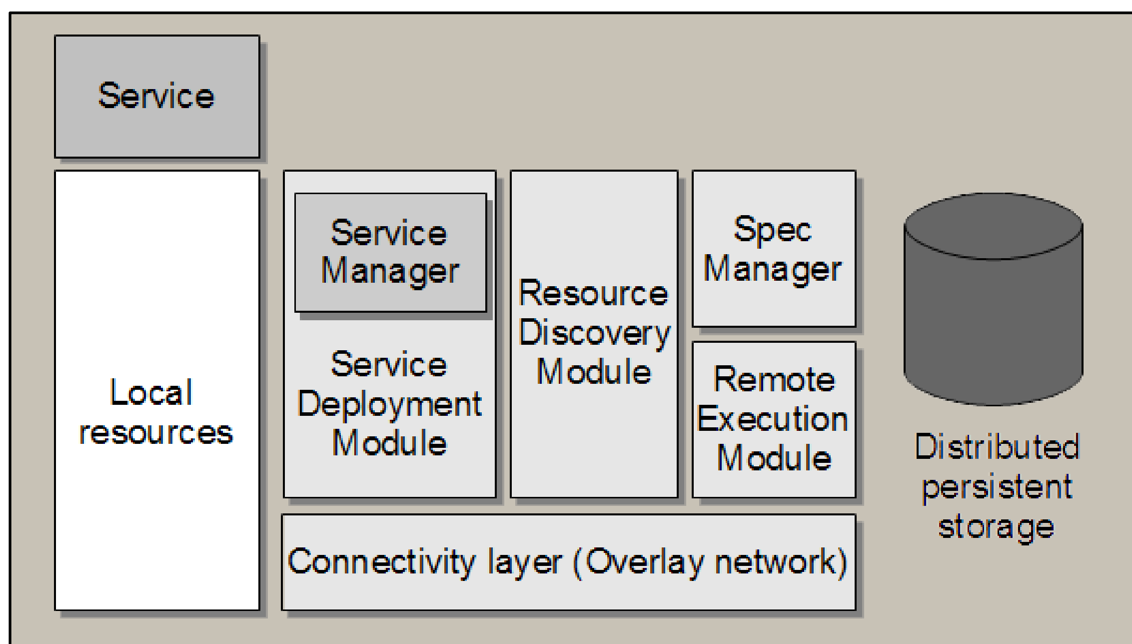
Des del punt de vista d'una aplicació, CoDeS és un directori de serveis (*SD*). Els usuaris poden registrar un servei en el directori, proporcionant-li una especificació. Després de registrar-lo, aquests serveis es poden desplegar, i CoDeS consulta l'especificació per a comprovar el nombre de rèpliques de cada servei que es requereixen. Una aplicació pot realitzar una consulta al directori (SD) indicant l'identificador d'un servei, per tal d'obtenir una llista de localitzacions del servei. Amb aquestes localitzacions, l'aplicació pot accedir directament a una instància del servei.



Overview de CoDeS (font: dpcs.uoc.edu)

[24] Third-party software: Component de software desenvolupat per a ser distribuït i utilitzat lliurement per al desenvolupament d'altre software derivat.

L'arquitectura de CoDeS és altament modular, per a facilitar que diferents implementacions dels mòduls es puguin connectar per a modificar o millorar el comportament de CoDeS o els mecanismes que utilitza. D'aquesta manera, CoDeS pot aprofitar els avenços en àrees com poden ser la computació *peer-to-peer*, l'emmagatzematge col·laboratiu, l'enrutament segur...



Arquitectura de CoDeS (font: dpcs.uoc.edu)

Els mòduls de CoDeS, com podem observar a la gràfica, són els següents:

- **Service Deployment Module (SDM).** El módul de desplegament de serveis és la part principal de CoDeS. Controla on els serveis s'executen. Alguns nodes mantenen desplegat un **Service Manager** (Controlador de serveis) o SM, un agent que s'encarrega de mantenir els serveis disponibles, seleccionant en quins nodes s'han d'executar i monitoritzant-los. Hi ha un determinat nombre de SM's, que els clients poden contactar per a localitzar serveis.
- **Resource Discovery Module (RDM).** El mòdul de descobriment de recursos troba quins nodes compleixen els requeriments necessaris per a executar un determinat servei.
- **Local resources**. Els recursos locals, que s'utilitzen per a executar serveis. En la implementació actual, CoDeS agafa els recursos de la màquina on s'està executant. Una manera segura i efectiva d'acotar i blindar els recursos que

hom vol cedir a la comunitat és utilitzant la virtualització, executant CoDeS dins d'una màquina virtual on els recursos que consumeix han sigut degudament limitats.

- **Connectivity layer.** La capa de connectivitat ofereix la funcionalitat de *Key-Based Routing*[25] (KBR), per a permetre una comunicació escalable entre nodes. Qualsevol implementació d'una altra capa de xarxa feta amb la interfície KBR, donat que CoDeS només utilitza operacions estàndard de KBR.
- **Spec Manager**. El gestor d'especificacions es fa servir per a interpretar les especificacions dels nodes i els requeriments dels serveis. Es poden utilitzar diferents implementacions per tal d'adaptar el sistema a diferents estàndards d'especificació.
- **Remote Execution Module.** El módul d'execució remota s'encarrega de controlar l'ús de recursos remots per als SM. Monitoritza quins nodes han sigut assignats per a dur a terme una tasca, i informen al SM en cas que hi hagi algun problema.
- **Distributed persistent storage**. L'emmagatzematge persistent distribuït es fa servir per a emmagatzemar tots els arxius requerits per a desplegar serveis (arxius executables, arxius de dades, arxius de descripció...). La implementació per defecte de CoDeS utilitza una DHT[26] construïda a sobre de la capa de connectivitat.

Overview *dels serveis de CoDeS*

Com veiem, el software que s'executa a CoDeS ho fa en forma de serveis. Cal que veiem com es programen aquests serveis, quines operacions i interfícies tenen associades, i com es distribueixen per la xarxa.

Un servei, en la seva essència, no és més que una classe de java que implementa la interfície *Service*, continguda dins de la llibreria de CoDeS. Aquesta interfície conté els següents mètodes que s'han de sobrecarregar:

- **deliverMessage**: és la funció que es crida en el servei quan s'envia un missatge al servei mitjançant l'operació de la API. Un dels paràmetres d'entrada és el missatge que s'ha enviat.

[25] KBR: sistema que permet trobar el node que es troba a la menor distància (mesurada en salts) d'un altre.
[26] DHT: *Distributed Hash Table*. Implementa un sistema distribuït i descentralitzat que resol les localitzacions d'una manera molt similar a un diccionari: emmagatzemant un conjunt de clau i valor.

- **start**: és la funció que es crida en el servei quan aquest es desplega.
- **stop**: és la funció que es crida en el servei quan aquest es para.

Registre d'un servei

Pel que fa a la difusió del servei, aquest es distribueix com a un arxiu comprimit en format *zip*[27]. Quan CoDeS registra un servei, se li ha d'indicar mitjançant una especificació com està estructurat aquest arxiu comprimit. Quan CoDeS el desplega, descomprimeix l'arxiu i navega entre els arxius resultants seguint el contingut de l'especificació que li passem.

La major part de l'especificació consisteix en un arxiu XML que es passa comprimit dins del servei. L'estructura d'aquest arxiu és similar a aquesta:

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <Execution>
  - <service>
      <sPath>/ServiceEcho/bin/</sPath>
      <sName>ServiceEcho.ServiceCEcho</sName>
    </service>
  - <classpath>
    - <lib>
        <lName>ServiceEcho/lib/connect.jar</lName>
      </lib>
    - <lib>
        <lName>ServiceEcho/lib/SyncBuffer.jar</lName>
      </lib>
    - <lib>
        <lName>ParserProva/lib/codes.jar</lName>
      </lib>
    </classpath>
  - <files>
    - <file>
        <name>ParserProva.zip</name>
        <format>zip</format>
      </file>
    </files>
  </Execution>
```

Especificació de serveis (font: Daniel Làzaro)

[27] zip: format de compressió d'arxius. És actualment un dels formats més coneguts i utilitzats, i hi ha moltes eines en diferents plataformes que l'implementen, com ara *Winzip*, *7zip*, etcétera.

El significat dels camps és:

- **spath:** ruta relativa (dins del zip) on es troben els binaris (*.class*) del servei.
- **sName:** nom de la classe que implementa la interfície Servei (amb la jerarquia de paquets inclosa)
- **lName:** ruta relativa (dins del zip) on es troba cada una de les llibreries que fa servir el servei. Per a garantir que una llibreria que un servei utilitza estigui present al node on s'executarà el servei, aquesta s'ha d'empaquetar dins del zip i ser especificada en aquesta secció.
- **name (dins el tag file):** nom de cada arxiu comprimit que hi hagi dins del zip que requereixin descompressió.
- **format (dins del tag file):** format en el que es troba comprimit l'arxiu en qüestió.

Quan es registra un servei a CoDeS, cal indicar de quin arxiu agafa aquesta especificació en XML. Per a tal efecte, es fa servir una instància de la classe *ServiceSpecification*, inclosa dins la llibreria de CoDeS. Un cop s'ha creat una instància d'aquesta classe, cal executar les següents funcions:

- **setExecSpec:** cal passar com a paràmetre un string amb un determinat format, que conté els següents string separats per dos punts (:). Identificador del servei, nom de l'arxiu .zip amb el que es distribuirá el servei, ruta relativa dins del zip de l'especificació xml. Per exemple, el valor d'aquest camp podria ser: "ServEcho:SrvEcho.zip:SpecificationECHO.xml".
- **setManagerSpec:** cal passar com a paràmetre un string contenint els paràmetres que necessita el SM per a saber com desplegar el servei. En la implementació actual de CoDeS, aquest consta simplement d'un nombre natural més gran que zero que indica quantes instàncies del servei volem que s'executin. Per exemple, "2".
- **setRequirements:** cal passar com a paràmetre un string contenint els paràmetres que necessita el RDM per a saber quins requeriments té el servei.

Un cop s'han executat correctament aquestes crides, ja es pot registrar el servei mitjançant la funció *registerService* de l'API de CoDeS. Cal passar dos paràmetres: un string que indiqui l'identificador de servei amb el que volem poder localitzar el servei (el mateix que hem passat en la crida *setExecSpec* de l'especificació) i la instància de *ServiceSpecification* que hem creat.

Upload d'un servei

Per tal que el servei que hem registrat es pugui difondre per tota la xarxa de CoDeS, cal que el copiem a la capa de dades. Per a tal efecte, cal que fem una crida a la funció *putFile* de l'API de CoDeS passant-li dos paràmetres: l'identificador que li volem donar a l'arxiu de servei, i la ruta de l'arxiu .zip del servei. Cal que l'arxiu .zip del servei estigui físicament present en la màquina on s'executa aquesta crida.

Desplegament d'un servei

Finalment, un cop el servei està registrat i el fitxer consta en la capa d'emmagatzemament, el podem desplegar cridant a la funció *deployService* de l'API de CoDeS. Cal passar-li un paràmetre, que es un String amb l'identificador del servei. Aquest identificador és únic per a cada recurs de CoDeS, i per tant no pot ser el mateix que hem facilitat al fer l'upload del fitxer. L'identificador que facilitem en aquest pas, serà l'identificador amb el que serà conegut el servei a CoDeS.

Localització d'un servei desplegat

Un cop ja tenim el servei desplegat a la xarxa, necessitem una manera de poder-lo localitzar a través del SDM. Per a fer-ho, cal cridar la funció *getLocationsSync* de l'api de CoDeS. Cal passar-li un paràmetre, que es un *String* indicant l'identificador del servei. El resultat d'aquesta funció és una llista amb els *ServiceHandle*s de les instàncies del servei que busquem. Mitjançant aquests *Handle*s es pot obtenir informació, com ara l'adreça del node on s'està executant, o també es poden utilitzar per a enviar missatges al servei, que son tractats a la funció *deliverMessage* de la interfície *Service*.

Comunicació amb els serveis

Fins ara hem vist com funciona CoDeS a nivell de serveis. Tanmateix, el que ens interessa es saber com ens podem comunicar amb aquests serveis, per a poder fer-hi consultes i per a poder-hi emmagatzemar dades.

Tenim diverses maneres de comunicar-nos amb un servei de CoDeS. Les més comuns són les que veurem a continuació.

Opció 1: Enviament de missatge a través de l'API

CoDeS està preparat perquè els seus serveis puguin rebre missatges i disposa d'un mecanisme d'enviament de missatges molt senzill d'utilitzar. A la part de l'aplicació, només cal cridar la funció sendServiceMessage de l'API de CoDeS passant

com a paràmetre el missatge que volem enviar. A la part del servei, cal sobrecarregar la funció *deliverMessage* perque tracti el missatge que s'ha enviat i executi el codi que nosaltres desitgem.

Aquest mecanisme té un problema: està pensat per a **comunicacions unidireccionals**, i per tant no tenim cap manera d'efectuar un retorn des del servei cap a l'aplicació. Així doncs, convé només utilitzar aquesta opció quan necessitem efectuar operacions que no requereixin retornar un resultat.

Opció 2: Apertura de socket en el desplegament

Per a la immensa majoria d'operacions del nostre sistema, necessitem que el servei retorni informació cap a l'aplicació. Així doncs, necessitem alguna via de comunicació bidireccional de dades.

La manera més senzilla de dur-ho a terme es obrint una via de comunicació punt a punt mitjançant *sockets*[28] de Java. Aquesta via de comunicació s'obre al desplegar el servei, mitjançant la implementació de la funció *start* de la interfície *Services*, i obre un *ServerSocket* en un port determinat conegut. Quan una aplicació es vol comunicar amb un servei, el procediment que ha de seguir és:

1. Localitzar el servei amb el que es vol comunicar
2. Obtenir l'adreça IP de node en el que s'executa, a través del seu *NodeHandle*
3. Obrir un *Socket* contra l'adreça del node i el port que aquest utilitza per a la comunicació
4. Comunicar la informació. La via de dades es bidireccional, i per tant podrà rebre respostes.

Cal tenir en compte que aquesta manera de comunicació té un inconvenient, que en el cas de HorNet és força important. Quan un servei es desplega, ha d'obrir un port TCP per a la connexió del socket, i aquest ha de ser conegut. Cada un dels serveis que es desplega ha de tenir un port diferent, donat que si no fos així es podrien produir conflictes, i el nombre de ports que es despleguen no son infinits donat que els ports públics s'identifiquen amb un nombre que va de 1000 a 216, i hi ha moltes més aplicacions utilitzant aquests ports en un ordinador (entre elles, el propi CoDeS!). A més, com que el port s'obriria en el moment de desplegar el servei i es tancaria en el moment de parar-lo, significaria que estem consumint un port per servei desplegat

[28] Socket: classe de java que implementa un protocol de comunicació punt a punt entre dos nodes.

durant tot el temps d'execució d'un programa, el qual ens limitaria el nombre d'usuaris que es poden registrar al sistema.

Opció 3: Missatge + Socket

A HorNet, apliquem una solució alternativa a les dues que s'han presentat ja, que no és més que una simple combinació d'aquestes. Hem optat per realitzar una comunicació bidireccional, utilitzant en una direcció una comunicació mitjançant missatge, i en una altra direcció una comunicació mitjançant socket.

El procés que es realitza és el següent

1. Localitzar el servei amb el que es vol comunicar.
2. Un cop coneixem el seu *handle*, enviem un missatge al servei mitjançant una crida a la API de CoDeS.
3. En la part del servei, hem d'obrir un *SocketServer*.
4. En la implementació de la funció *deliverMessage* del servei, hem implementat el tractament que s'ha de realitzar al missatge rebut.
5. Un cop tenim la resposta, obrim un *Socket* en direcció a l'aplicació, on hem obert un port en el pas 3. Les dades de connexió (adreça IP i port) s'han d'haver inclós en el cos del missatge que hem enviat. Transmetem la resposta i tanquem la comunicació.

Aquest tipus de comunicació és una mica més complex en la seva execució, però té avantatges respecte a ambdues alternatives. Respecte a la primera, disposem d'una comunicació bidireccional; respecte a la segona, no necessitem assignar números de port a cada un dels molts serveis que es desplegaran en el sistema, ni tenir connexions a *sockets* obertes tota la estona: l'aplicació és la que s'encarrega d'assignar un número de port de dins d'un *pool*[29] (que pot ser molt reduït, de tants com connexions simultànies volem permetre sense bloquejar-se) i les connexions a aquests ports s'obren i es tanquen a demanda, només quan és necessari.

Per a implementar aquesta opció hem necessitat dues classes addicionals: la classe *PortPool* i la classe *DeliveryMessage*. La implementació d'aquesta opció la hem realitzat en una classe abstracta que sobrecarregaran les aplicacions que vulguin utilitzar els serveis de HorNet: la classe *ServiceInvokingApp*.

[29] pool: Conjunt de dades inicialitzables disposades per a la seva utilització.

Classe PortPool

Hem vist que és possible implementar una comunicació híbrida que ens permeti obrir fer transaccions bidireccionals entre aplicació i servei. Tanmateix, un dels requeriments d'aquesta solució és que l'aplicació és la que ha de gestionar els ports, per tal que mai es produeixin conflictes entre aplicacions que estan esperant un missatge. És per això que hem d'implementar una classe que ens proporcioni mètodes per a controlar aquests ports necessaris per a la comunicació. Hem anomenat a aquesta classe *PortPool*.

PortPool és un *singleton* que estarà instanciat a la màquina on s'executa l'aplicació. Els serveis, per tant, no hi tindran accés. Les operacions que es poden executar en aquesta classe són les següents:

- **getInstance** (). Retorna la instància de *PortPool*. En cas que no existeixi, en crea una de nova, contenint els ports del 15400 al 15499.
- **getPort** (). Retorna un enter corresponent al primer port lliure del *pool*, i el marca com a ocupat per a que ningú més el pugui agafar.
- **returnPort** (int p). Indica que el port *p* ja torna a estar lliure.
- **generateID** (String str). Retorna un string, una cadena de vuit caràcters corresponent a un identificador generat a partir del string str. Aquest identificador té un component seqüencial, i per tant dues crides successives a la funció *generateID* amb el mateix paràmetre d'entrada produeixen identificadors diferents.

La manera d'utilitzar aquesta classe per a controlar quin port utilitzar és la següent:

1. Obtenir la instància de PortPool mitjançant el mètode estàtic getInstance.
2. Reservar un port mitjançant el mètode *getPort*.
3. (Opcional) Generar un identificador per a la petició mitjançant el mètode *generateID* .
4. Enviar un missatge al servei mitjançant el mètode *sendServiceMessage* de l'API de CoDeS, indicant l'adreça del node on s'executa l'aplicació i el port que acabem de reservar. Si hem executat el pas 3, també podem enviar l'identificador de petició.
5. Obrir un *SocketServer* que escolti connexions entrants.

6. Quan es produeixi una connexió, rebre les dades que es transmetin. Aquestes dades son la resposta que estavem esperant del servei.
7. Tancar la connexió.
8. Retornar el port que hem utilitzat a *PortPool* mitjançant el mètode *returnPort*.
9. (Opcional) Comprovar que la resposta es correspon a la petició, comparant l'identificador de petició que hem enviat amb el que hem rebut.

Donat que la resposta s'envia a través d'un *Socket*, aquesta sempre serà un string. Depenent quina hagi sigut la operació que hagi originat la resposta, caldrà interpretar-lo d'una manera o d'una altra.

Classe DeliveryMessage

Ara que ja tenim una manera de controlar els ports que utilitzem per a la comunicació de manera centralitzada des de l'aplicació, necessitem una manera de comunicar aquestes dades al servei que les ha d'utilitzar. Ens trobem que la classe *ServiceMessage* que es passa per paràmetres al invocar la funció *serviceMessage* és un missatge sense cos, que només té un identificador i prou. Per tant, ens caldrà dissenyar una superclasse de *ServiceMessage* que ens permeti passar-li la informació que volem.

La classe *DeliveryMessage* no conté cap mètode (a banda dels mètodes heretats de la superclasse): es tracta d'un simple tipus abstracte de dades que emmagatzema la informació necessària per a fer una crida a una operació dels serveis de HorNet. Conté els següents camps:

Paràmetre	Tipus de dades	Descripció
operation	String	Tipus d'operació que volem utilitzar
parameter1	Object	Primer paràmetre de la operació
parameter2	Object	Segon paràmetre de la operació
parameter3	Objec	Tercer paràmetre de la operació
retAddress	String	Adreça IP del node que ha sol·licitat el missatge
retPort	int	Port TCP que escoltarà la resposta
requestId	String	Identificador de la petició.

Cal tenir en compte una cosa important: tots els atributs de la classe han de poder ser serialitzables, donat que aquesta classe s'ha de poder empaquetar per a

viatjar en una crida RMI[30]. Els atributs que tenen un tipus concret ja són serialitzables, però per als atributs que són de la classe abstracta *Object*, cal tenir en compte aquesta consideració. És per això que totes les classes susceptibles de ser usades en una crida a servei que hem implementat a HorNet, tenen funcions per a convertir a *String* i per a realitzar el pas invers.

A més, també cal considerar una petita limitació tecnològica del RMI: a tots els nodes on es desplegui HorNet, la classe DeliveryMessage ha de figurar al classpath. Si no és així, si es realitza una crida a funció en un servei de HorNet desplegat en un node on no existeix la classe DeliverMessage, aquest retornarà un error de classe no trobada. En realitzar el procés de *unmarshalling*[31], el RMI necessita tenir el binari de la classe que es passa per paràmetres al mètode *sendMessage*. Cal tenir en compte aquesta limitació de cara a saber com difondre l'aplicació.

Classe ServiceInvokingApp

Com hem explicat abans, hem implementat el protocol de comunicació de HorNet en la classe abstracta *ServiceInvokingApp*. Aquesta classe disposa dels següents mètodes:

- **registerService** (int id, String service). Registra el servei especificat pels paràmetres, on *id* és l'identificador d'un usuari i *service* és un d'aquests valors: "Messages", "Timeline", "Followers", "Following", "Profile" o "Mentions". Retorna un string, corresponent a l'identificador del servei.
- **registerIndexService** (). Registra el servei especial d'indexació, i en retorna el seu identificador de servei.
- **registerAuthService** (). Registra el servei especial d'autenticació, i en retorna el seu identificador de servei.
- **callServiceOperation** (String sid, String operation, Object param1). Crida la operació determinada pel paràmetre *operation* en el servei amb identificador *sid*, amb el paràmetre *param1*. Hi dues versions més d'aquesta crida, per a passar fins a tres paràmetres. Aquesta funció obre un *socket* amb un *timeout* [32]de 10 segons, i retorna el *string* de resposta.

[30] RMI: Remote Method Invocation, un mecanisme ofert per Java per a realitzar crides remotes.

[31] unmasrhall: procés de deserialització d'una crida realitzada remotament, per tal que aquesta es pugui executar en el node on ha estat cridada.

[32] timeout: error produit per una operació que ha trigat massa temps en executar-se

- **callThreadedServiceOperation** (String sid, String operation, Object param1). Crida la operació determinada pel paràmetre *operation* en el servei amb identificador *sid*, amb el paràmetre *param1*. Hi dues versions més d'aquesta crida, per a passar fins a tres paràmetres. A diferència de l'anterior crida, l'aplicació no es bloqueja fins que es rep la resposta sinó que es segueix executant, i en arribar la resposta es descarta.

Tot i que és transparent per a l'aplicació, cal tenir en compte que, en la implementació actual de la classe, la funció de *callServiceOperation* efectua la comprovació d'identificador de petició. En cas que els identificadors no es corresponguin, i per tant ens trobem davant d'un conflicte provocat per ports creuats, la resposta serà un codi d'error.

Cal entendre bé la diferència entre les funcions *callServiceOperation* i *callThreadedServiceOperation*. Caldrà fer servir la primera en operacions de *get* on necessitem conèixer la resposta, o bé en operacions de *put* que requereixin un cert ordre per a executar-se correctament. En canvi, la segona és ideal per a operacions de *put* que es puguin executar en paral·lel, com per exemple l'actualització del servei *Timeline* dels seguidors d'un usuari que acaba de publicar un missatge.

Especificació dels serveis

Un cop especificada la manera en la que es realitzaran les crides a serveis, cal que especifiquem els serveis que hem dissenyat, tenint en compte la implementació que hem decidit aplicar i a la tecnologia que estem utilitzant.

Hem optat per comunicar-nos amb els serveis mitjançant l'enviament de missatges, i que aquests responguin a l'aplicació enviant un flux de caràcters mitjançant un *socket*. Així doncs, podem considerar que la implementació de la funció *deliverMessage* de cada servei consistirà en el següent:

1. Descodificar el missatge rebut (de tipus *DeliveryMessage*).
2. Fer unes accions o unes altres segons quin sigui el contingut de l'atribut *operation* del missatge.
3. Construir i enviar la resposta.

Per tant, per a cada servei necessitem saber quins són els possibles valors de l'atribut *operation*, quines accions efectuarà el servei per a cada valor, i també quins possibles valors pot tenir la resposta que es generarà.

Servei User.Messages

Aquest servei ha sigut dissenyat per a emmagatzemar totes les entrades de missatges relacionades amb un usuari determinat. En el disseny es va mencionar que necessitàvem una estructura de dades per a emmagatzemar aquesta informació; aquesta estructura de dades és una classe de *java* anomenada *MsgEntry*. Aquesta classe té els següents atributs:

Atribut	Tipus de dades	Descripció
ts	int	Identificador del missatge dins del servei
userId	int	Identificador de l'usuari que publica el missatge. Es tracta d'un camp redundant: tots els missatges que s'emmagatzemen en aquest servei tenen el mateix usuari.
origuser	int	Identificador de l'autor original del missatge
message	String	Cos del missatge
date	long	Data de publicació del missatge

A més, disposa dels següents mètodes públics:

- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *MsgEntry* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatejat, retorna una instància de la classe *MsgEntry.*

Finalment, cal que veiem com es comporta la funció *deliverMessage* del servei segons quin sigui el valor del missatge de tipus *DeliveryMessage* que rep.

Afegir missatges

Codi d'operació	#Param	Tipus	Descripció
AddMessage	1	String	Missatge a afegir
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb el mètode estàtic *MsgEntry.parse()* i afegeix el missatge al sistema amb un *ts* nou.		
String resultat	Retorna el *timestamp* del missatge un cop afegit al sistema		

Eliminar missatges

Codi d'operació	#Param	Tipus	Descripció
RemoveMessage	1	Integer	*ts* del missatge a eliminar
	2	-	-
Procediment	Obté un int amb la funció *intValue* del paràmetre 1 i elimina el missatge del sistema.		
String resultat	Retorna "OK".		

Obtenir tots els missatges

Codi d'operació	#Param	Tipus	Descripció
RemoveMessage	1	-	-
	2	-	-
Procediment	Cerca tots els missatges guardats al sistema.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir nombre de missatges

Codi d'operació	#Param	Tipus	Descripció
GetMessageCount	1	-	-
	2	-	-
Procediment	Compta els missatges guardats al sistema		
String resultat	Retorna el nombre de missatges.		

Obtenir missatges segons TS

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanTS	1	Integer	Timestamp de referència
	2	-	-
Procediment	Obté un *int* a través de la funció *intValue* del paràmetre 1, i retorna tots els missatges que tenen un *ts* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir missatges segons Data

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanDate	1	Long	Data de referència
	2	-	-
Procediment	Obté un *long* a través de la funció *longValue* del paràmetre 1, i retorna tots els missatges que tenen un *date* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Servei User.Timeline

De manera similar al servei *User.Messages*, el servei *User.Timeline* també fa servir un tipus abstracte de dades per a manipular la informació que emmagatzema. Aquest tipus de dades es troba implementat a la classe *TlEntry*.

Atribut	Tipus de dades	Descripció
ts	int	Identificador del missatge dins del servei
origTS	int	Identificador del missatge dins del servei *Messages* de l'autor.
userid	int	Identificador de l'usuari que publica el missatge
origuser	int	Identificador de l'autor original del missatge
message	String	Cos del missatge
date	long	Data de publicació del missatge

A més, disposa dels següents mètodes públics:

- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *TlEntry* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatejat, retorna una instància de la classe *TlEntry.*

Finalment, cal que veiem com es comporta la funció *deliverMessage* del servei segons quin sigui el valor del missatge de tipus *DeliveryMessage* que rep.

Afegir missatges

Codi d'operació	#Param	Tipus	Descripció
AddMessage	1	String	Missatge a afegir
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb el mètode estàtic *TlEntry.parse()* i afegeix el missatge al sistema amb un *ts* nou.		
String resultat	Retorna "OK".		

Eliminar missatges

Codi d'operació	#Param	Tipus	Descripció
RemoveMessage	1	Integer	*ts* del missatge a eliminar
	2	-	-
Procediment	Obté un int amb la funció *intValue* del paràmetre 1 i elimina el missatge del sistema.		
String resultat	Retorna "OK".		

Obtenir tots els missatges

Codi d'operació	#Param	Tipus	Descripció
GetAllMessages	1	-	-
	2	-	-
Procediment	Cerca tots els missatges guardats al sistema.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir els X usuaris més recents

Codi d'operació	#Param	Tipus	Descripció
GetXNewerItems	1	Integer	Nombre d'usuaris a retornar
	2	-	-
Procediment	Obté un *int* amb la funció *intValue* del paràmetre 1 i retorna aquell nombre de missatges, sent els més recents del servei.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir els usuaris entre X i Y

Codi d'operació	#Param	Tipus	Descripció
GetFromXToY	1	Integer	Cota superior X
	2	Integer	Cota inferior Y
Procediment	Assignant a cada missatge un nombre, sent "1" el missatge més nou, retorna els missatges entre "X" i "Y".		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir missatges segons TS

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanTS	1	Integer	Timestamp de referència
	2	-	-
Procediment	Obté un *int* a través de la funció *intValue* del paràmetre 1, i retorna tots els missatges que tenen un *ts* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir missatges segons Data

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanDate	1	Long	Data de referència
	2	-	-
Procediment	Obté un *long* a través de la funció *longValue* del paràmetre 1, i retorna tots els missatges que tenen un *date* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Servei User.Followers

El servei *User.Followers* no tracta amb estructures de dades, sinó que només s'encarrega d'emmagatzemar una referència cap als usuaris que es troben representats en el servei. Com que els usuaris es poden identificar mitjançant el seu *id*, només ens cal que el servei *User.Followers* sigui capaç de gestionar llistes d'enters.

Aquest és el comportament de la funció *deliverMessage* del servei segons quin sigui el valor del missatge de tipus *DeliveryMessage* que rep.

Afegir usuari

Codi d'operació	#Param	Tipus	Descripció
AddUser	1	Integer	Usuari a afegir
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i l'afegeix al servei. Si l'usuari existia, no fa res.		
String resultat	Retorna "OK".		

Eliminar usuari

Codi d'operació	#Param	Tipus	Descripció
RemoveUser	1	Integer	Usuari a esborrar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i l'elimina del servei. Si l'usuari no existia, no fa res.		
String resultat	Retorna "OK".		

Existeix usuari

Codi d'operació	#Param	Tipus	Descripció
ContainsUser	1	Integer	Usuari a comprovar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i comprova si aquest existeix al servei.		
String resultat	Retorna "Y" si l'usuari existeix, i "N" altrament.		

Obtenir nombre d'usuaris

Codi d'operació	#Param	Tipus	Descripció
GetUserCount	1	-	-
	2	-	-
Procediment	Consulta al servei quants usuaris hi ha.		
String resultat	Retorna el nombre d'usuaris existents al servei.		

Obtenir tots els usuaris

Codi d'operació	#Param	Tipus	Descripció
GetAllUsers	1	-	-
	2	-	-
Procediment	Consulta al servei la llista d'usuaris que té emmagatzemats.		
String resultat	Retorna els identificadors dels usuaris, separats per salts de línea.		

Obtenir usuaris més recents

Codi d'operació	#Param	Tipus	Descripció
GetXNewerUsers	1	Integer	Nombre d'usuaris a retornar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i retorna com a màxim aquell nombre d'usuaris, començant pel més recent.		
String resultat	Retorna els identificadors dels usuaris, separats per salts de línea.		

Servei User.Following

De forma similar al que vam veure en el disseny, el comportament d'aquest servei és exactament igual al de *User.Followers*, i la diferència entre ambdós serveis és purament conceptual. Així doncs, el comportament de la funció *deliverMessage* segons el valor de l'objecte *DeliveryMessage* que li passem com a paràmetre serà el següent:

Afegir usuari

Codi d'operació	#Param	Tipus	Descripció
AddUser	1	Integer	Usuari a afegir
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i l'afegeix al servei. Si l'usuari existia, no fa res.		
String resultat	Retorna "OK".		

Eliminar usuari

Codi d'operació	#Param	Tipus	Descripció
RemoveUser	1	Integer	Usuari a esborrar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i l'elimina del servei. Si l'usuari no existia, no fa res.		
String resultat	Retorna "OK".		

Existeix usuari

Codi d'operació	#Param	Tipus	Descripció
ContainsUser	1	Integer	Usuari a comprovar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i comprova si aquest existeix al servei.		
String resultat	Retorna "Y" si l'usuari existeix, i "N" altrament.		

Obtenir nombre d'usuaris

Codi d'operació	#Param	Tipus	Descripció
GetUserCount	1	-	-
	2	-	-
Procediment	Consulta al servei quants usuaris hi ha.		
String resultat	Retorna el nombre d'usuaris existents al servei.		

Obtenir tots els usuaris

Codi d'operació	#Param	Tipus	Descripció
GetAllUsers	1	-	-
	2	-	-
Procediment	Consulta al servei la llista d'usuaris que té emmagatzemats.		
String resultat	Retorna els identificadors dels usuaris, separats per salts de línea.		

Obtenir usuaris més recents

Codi d'operació	#Param	Tipus	Descripció
GetXNewerUsers	1	Integer	Nombre d'usuaris a retornar
	2	-	-
Procediment	Obté un *int* a partir de la comanda *intValue* del paràmetre 1 i retorna com a màxim aquell nombre d'usuaris, començant pel més recent.		
String resultat	Retorna els identificadors dels usuaris, separats per salts de línea.		

Servei User.Mentions

El servei *User.Mentions* es comporta gairebé de manera idèntica al servei *User.Timeline*, la diferència entre ambdós serveis és purament conceptual. Tanmateix, l'estructura de dades que necessita per a treballar correctament és lleugerament diferent, i es troba programada a la classe *MentEntry*. La classe té els següents atributs:

Atribut	Tipus de dades	Descripció
ts	int	Identificador del missatge dins del servei
origTS	int	Identificador del missatge dins del servei *Messages* de l'autor.
userid	int	Identificador de l'usuari que publica el missatge
origuser	int	Identificador de l'autor original del missatge
message	String	Cos del missatge
date	long	Data de publicació del missatge

A més, disposa dels següents mètodes públics:

- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *MentEntry* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatat, retorna una instància de la classe *MentEntry.*

Finalment, cal que veiem com es comporta la funció *deliverMessage* del servei segons quin sigui el valor del missatge de tipus *DeliveryMessage* que rep.

Afegir missatges

Codi d'operació	#Param	Tipus	Descripció
AddMessage	1	String	Missatge a afegir
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb el mètode estàtic *MentEntry.parse()* i afegeix el missatge al sistema amb un *ts* nou.		
String resultat	Retorna "OK".		

Eliminar missatges

Codi d'operació	#Param	Tipus	Descripció
RemoveMessage	1	Integer	*ts* del missatge a eliminar
	2	-	-
Procediment	Obté un int amb la funció *intValue* del paràmetre 1 i elimina el missatge del sistema.		
String resultat	Retorna "OK".		

Obtenir tots els missatges

Codi d'operació	#Param	Tipus	Descripció
GetAllMessages	1	-	-
	2	-	-
Procediment	Cerca tots els missatges guardats al sistema.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir els X usuaris més recents

Codi d'operació	#Param	Tipus	Descripció
GetXNewerItems	1	Integer	Nombre d'usuaris a retornar
	2	-	-
Procediment	Obté un *int* amb la funció *intValue* del paràmetre 1 i retorna aquell nombre de missatges, sent els més recents del servei.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir els usuaris entre X i Y

Codi d'operació	#Param	Tipus	Descripció
GetFromXToY	1	Integer	Cota superior X
	2	Integer	Cota inferior Y
Procediment	Assignant a cada missatge un nombre, sent "1" el missatge més nou, retorna els missatges entre "X" i "Y".		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir missatges segons TS

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanTS	1	Integer	Timestamp de referència
	2	-	-
Procediment	Obté un *int* a través de la funció *intValue* del paràmetre 1, i retorna tots els missatges que tenen un *ts* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Obtenir missatges segons Data

Codi d'operació	#Param	Tipus	Descripció
GetItemsNewerThanDate	1	Long	Data de referència
	2	-	-
Procediment	Obté un *long* a través de la funció *longValue* del paràmetre 1, i retorna tots els missatges que tenen un *date* més gran.		
String resultat	Retorna la llista de missatges, separats per salts de línea.		

Servei User.Profile

Tot i que el servei *User.Profile* no tracta amb llistes de dades sinó que ho fa amb una sola instància, el tipus de dades amb el que treballa és bastant complex. Per tant, l'hem implementat en una classe anomenada *User* que té els següents atributs:

Atribut	Tipus de dades	Descripció
uid	int	Identificador numèric de l'usuari
username	String	Nom de l'usuari, identificador informal
avatar	String	URL contenint l'avatar de l'usuari
published	Boolean	Val "cert" si l'usuari accedeix a fer el seu perfil públic, "fals" altrament.
properties	java.util.Vector	Llista els noms dels paràmetres opcionals del perfil
values	java.util.HashMap	Permet emmagatzemar els valors dels paràmetres opcionals del perfil en parelles <nom, valor>.
indPublish	java.util.Vector	Indica, per a cada paràmetre opcional, si l'usuari accedeix a fer-lo públic.

A més, disposa dels següents mètodes públics:

- ***getters / setters***. A diferència de les classes associades als altres serveis, la classe User té els atributs privats. Això significa que per accedir a la majoria d'ells calen operacions de get i set. Aquestes operacions són les següents:
  - getID
  - setID

- getUsername
- setUsername
- getAvatar
- setAvatar
- isPublished
- setPublished
- getAllProperties

- **getProperty** (String prop). Retorna el contingut del camp *values* corresponent a la propietat *prop*.
- **setProperty** (String key, String value). Afegeix *key* a la llista *properties*, i afegeix un parell (*key*, *value*) al diccionari *values*.
- **getIndivPub** (int i). Retorna el valor de la posició *i* de la variable *indPublish*. Els valors de *indPublish* i de *properties* van emparellats per posició.
- **addIndivPub** (String pub). Afegeix *pub* a la llista *indPublish*.
- **flushAllProperties** (). Elimina el contingut de les variables *properties*, *values* i *indPublish*.
- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *User* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatejat, retorna una instància de la classe *User.*

Finalment, el comportament de la funció *deliverMessage* del servei segons el valor del missatge de tipus *DeliveryMessage* és el següent:

Obtenir perfil

Codi d'operació	#Param	Tipus	Descripció
GetProfile	1	-	-
	2	-	-
Procediment	Obté el perfil emmagatzemat en el servei.		
String resultat	Retorna el perfil serialitzat.		

Actualitzar perfil

Codi d'operació	#Param	Tipus	Descripció
PutProfile	1	String	Perfil d'usuari serialitzat
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb la operació estàtica *User.parse* i sobreescriu el perfil emmagatzemat al servei.		
String resultat	Retorna "OK".		

Servei Authentication

El servei *Authentication* també utilitza un tipus abstracte de dades per a gestionar les llistes d'informació que manté. Hem implementat aquest tipus de dades en una classe anomenada *AuthEntry* que té els següents paràmetres:

Atribut	Tipus de dades	Descripció
userid	int	Identificador numèric de l'usuari
name	String	Nom de l'usuari, identificador informal
pass	String	Contrasenya de l'usuari

A més, disposa dels següents mètodes públics:

- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *AuthEntry* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatejat, retorna una instància de la classe *AuthEntry.*

El comportament de la funció *deliverMessage* segons el contingut de *DeliveryMessage* és el següent:

Afegir usuari

Codi d'operació	#Param	Tipus	Descripció
InsertUser	1	String	Usuari a afegir
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb el mètode estàtic *AuthEntry.parse()* i afegeix l'usuari al sistema. Si el nom d'usuari ja existeix en el servei i els identificadors numérics no es corresponen, retorna un error de clau primària.		
String resultat	Retorna "OK" si tot ha anat bé, o "Primary key infraction" si s'ha produït un error.		

Eliminar usuari

Codi d'operació	#Param	Tipus	Descripció
RemoveUser	1	Integer	*userid* de l'usuari a eliminar
	2	-	-
Procediment	Obté un int amb la funció *intValue* del paràmetre 1 i elimina l'usuari del servei.		
String resultat	Retorna "OK".		

Obtenir tots els usuaris

Codi d'operació	#Param	Tipus	Descripció
GetAllUsers	1	-	-
	2	-	-
Procediment	Cerca tots els usuaris guardats al sistema.		
String resultat	Retorna la llista d'usuaris en strings formatejats, separats per salts de línea.		

Obtenir usuari autenticat

Codi d'operació	#Param	Tipus	Descripció
GetUserAuth	1	String	Nom d'usuari
	2	String	Contrasenya
Procediment	Obté l'identificador numéric de l'usuari amb username indicat pel paràmetre 1, si i només si la contrasenya coincideix amb el paràmetre 2.		
String resultat	Retorna l'identificador de l'usuari, o "Login Failed" altrament.		

Obtenir id d'usuari

Codi d'operació	#Param	Tipus	Descripció
GetUserId	1	String	Nom d'usuari
	2	-	-
Procediment	Obté l'identificador numéric de l'usuari amb username indicat pel paràmetre 1		
String resultat	Retorna l'identificador de l'usuari, o "User does not exist" si l'usuari no figura en el servei.		

Obtenir següent usuari

Codi d'operació	#Param	Tipus	Descripció
GetNextId	1	-	-
	2	-	-
Procediment	Obté el següent identificador d'usuari assignable.		
String resultat	Retorna l'identificador d'usuari.		

Servei Indexation

La informació que de cerca que s'emmagatzema al sistema per a cada usuari es desa en un tipus abstracte de dades, que nosaltres hem implementat en una classe anomenada *IndexEntry*. Aquesta classe té els següents atributs:

Atribut	Tipus de dades	Descripció
id	int	Identificador numèric de l'usuari
properties	Vector	Informació de cerca relacionada amb l'usuari

A més, disposa dels següents mètodes públics:

- **serialize** (). Retorna un *string*, consistent en la serialització de la instància de *IndexEntry* sobre la que s'executa.
- **parse** (String s). És una funció estàtica, que es pot cridar sense instanciar la classe. Realitza el pas invers, donat un *string s* correctament formatejat, retorna una instància de la classe *IndexEntry.*

El comportament de la funció *deliverMessage* segons el valor de la classe *DeliveryMessage* és el següent:

Publicar informació

Codi d'operació	#Param	Tipus	Descripció
Publish	1	String	Usuari a afegir
	2	-	-
Procediment	Deserialitza el paràmetre 1 amb el mètode estàtic *IndexEntry.parse()* i afegeix l'usuari al sistema. Si l'identificador ja existia en el sistema, en sobreescriu la informació.		
String resultat	Retorna "OK"		

Cerca

Codi d'operació	#Param	Tipus	Descripció
Search	1	String	Patró de cerca
	2	-	-
Procediment	Cerca aparicions completes del paràmetre 1 dins del contingut de *properties* de cada usuari.		
String resultat	Retorna una llista amb els identificadors d'usuaris per al qual hi ha hagut coincidències, separats per salts de línea.		

Deixar de publicar informació

Codi d'operació	#Param	Tipus	Descripció
RemoveUser	1	Integer	Identificador numéric de l'usuari.
	2	-	-
Procediment	Obté un *int* amb el mètode *intValue* del paràmetre 1 i elimina l'entrada corresponent del sistema.		
String resultat	Retorna "OK"		

Persistència de les dades

Hem pogut veure com es gestiona la obtenció i el processament de les dades en els vuit tipus de servei que es despleguen a *HorNet*. Tanmateix, cal que decidim com

es gestionarà la persistència de les dades per tal que aquestes no es perdin i estiguin sempre disponibles.

Podem gestionar la persistència de dues maneres: en memòria o bé en disc.

Persistència en memòria

Dins de cada servei, es despleguen en forma de variables les estructures de dades necessàries per a mantenir accessibles totes les dades del servei. Es tracta d'una solució que facilitaria un accés molt ràpid, donat que les dades carregades en memòria es llegeixen molt més ràpidament que si es troben emmagatzemades a disc, però té dos problemes a tenir en compte:

- **Inconsistències**. Els serveis s'instancien diverses vegades en el sistema, per a garantir la seva disponibilitat. L'emmagatzemament de dades en memòria serà independent per a cada servei, i per tant caldrà que serveis del mateix tipus es comuniquin constantment per tal de que les dades siguin sempre consistents. Aquesta comunicació podria consumir molt temps i generar tràfic addicional a la xarxa, i en cas que aquesta comunicació per algun motiu no es pogués produir, podríem trobar-nos davant de situacions on el model de dades està corrupte o incomplet.
- **Volatilitat de les dades**. Si per algun motiu es produís una fallada del sistema, en la qual els nodes que estan allotjant un servei caiguessin, les dades que s'havien estat emmagatzemant en aquells serveis es perdien per sempre.

Persisténcia en disc

La opció per la que hem optat a HorNet és emmagatzemar les dades en el disc, per a garantir un nivell més de seguretat de cara a pèrdues d'informació. Per tal que tots els nodes que allotgin un servei puguin accedir lliurement a les dades, aquestes no es desen en el disc dur local, sinó que aprofiten la capa d'emmagatzematge de CoDeS per a tal efecte. Així doncs, el procediment que es du a terme és el següent:

1. Descarregar l'arxiu que s'encarrega de fer persistents les dades.
2. Obtenir la informació o fer els canvis pertinents.
3. Només en cas que l'arxiu s'hagi modificat, tornar a carregar-lo a la capa d'emmagatzematge.
4. Eliminar l'arxiu local per tal d'alliberar espai en disc del node.

El procediment és força més lent, però per una banda ens garanteix que les dades estaran disponibles passi el que passi amb els nodes, i per altra banda ens assegura que els nodes podran accedir concurrentment a les dades seguint el procediment que s'ha descrit a dalt, sense necessitar trànsit addicional per a sincronitzar-se.

La referència de serveis amb els arxius de dades que gestionen la persistència a HorNet és la següent:

Servei	Arxiu Associat
X.Timeline	X.Timeline.raw
X.Messages	X.Messages.raw
X.Followers	X.Followers.raw
X.Following	X.Following.raw
X.Mentions	X.Mentions.raw
X.Profile	X.Profile.xml
Indexation	Index.raw
Authentication	Auth.raw

Metodologia

Per a l'elaboració d'aquest projecte, he seguit una metodologia consistent en l'aplicació d'una sèrie de fases de manera seqüencial, entre les quals hi havia un cert grau de solapament. Per a aconseguir aquest grau de solapament, he ordenat les fases de manera que entre una fase i una altra hi hagués una certa afinitat. Per tant, la planificació d'aquestes fases i l'ordre en el que s'han aplicat és important.

Planificació

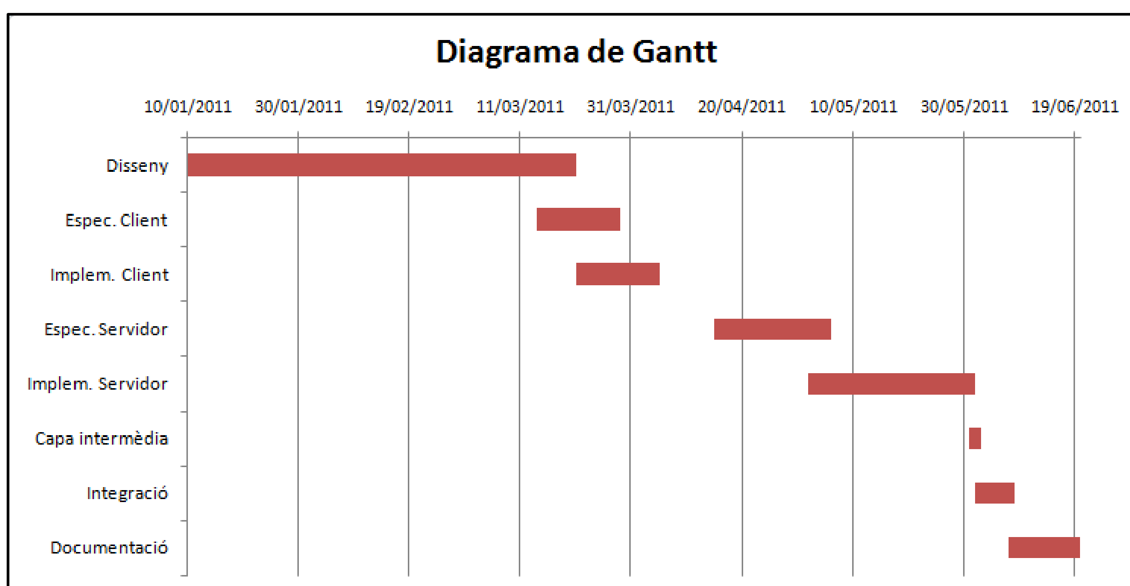
L'ordre de les fases i la seva descripció és el següent:

- Fase 1: **Disseny de l'aplicació**. Aquesta fase ha consistit en una sèrie de sessions de *brainstorming*[33] i la lectura d'articles a mode de documentació, que han acabat resultant en un document detallant les característiques de l'aplicació, i els aspectes que es podia considerar abraçar.
- Fase 2: **Especificació del client**. Aquesta fase ha consistit en l'estudi d'aplicacions similar a la que voliem desenvolupar, per a veure quins components ens podien ser útils de cara a dissenyar una interfície amigable.
- Fase 3: **Implementació i testing del client**. Aquesta fase ha consistit simplement en desenvolupar el client web a partir de les tecnologies que es van decidir en els requisits no funcionals. Per a poder implementar el client sense haver implementat el servidor, es va utilitzar una implementació de la classe *DataManager* que efectua el nexe entre client i servidor, per a que es connectés a una base de dades local, molt més senzilla d'implementar per a un entorn de proves d'interfície similar al resultat final.
- Fase 4: **Especificació del servidor**. Aquesta fase ha consistit en, a partir del disseny que s'havia elaborat en fase 1, aplicar els canvis necessaris considerant les tecnologies escollides i les seves limitacions per a obtenir una especificació dels serveis de CoDeS que voliem implementar.
- Fase 5: **Implementació i testing del servidor.** Servei a servei, s'ha dut a terme la implementació del servidor en aquesta fase. Pel que fa al testing, s'han realitzat tests individuals de cada servidor mitjançant petites subclasses de *ServiceInvokingApp* que han servit de jocs de proves.

[33] brainstorming: procés creatiu que consisteix en un grup de persones generant idees de manera massiva, per a després garbellar-les i adoptar les millors.

- Fase 6: **Implementació de la capa intermèdia.** En aquesta fase s'ha fet la implementació de la classe *CoDeSDataManager* per tal de disposar d'un nexe entre els dos components, funcionals per separat.
- Fase 7: **Integració.** En aquesta fase s'ha comprovat que tot funcionés bé després d'incorporar la classe *CoDeSDataManager* al client i de fer un desplegament de CoDeS. Com era d'esperar, no tot va anar bé, i no totes les funcionalitats que estaven contemplades en el disseny es van poder integrar correctament.
- Fase 8: **Documentació.** En aquesta fase s'ha recopilat la informació obtinguda en cada una de les fases, se li ha donat un format, i s'han generat els continguts addicionals necessaris per a la finalització de la memória: diagrames, taules, etcétera.

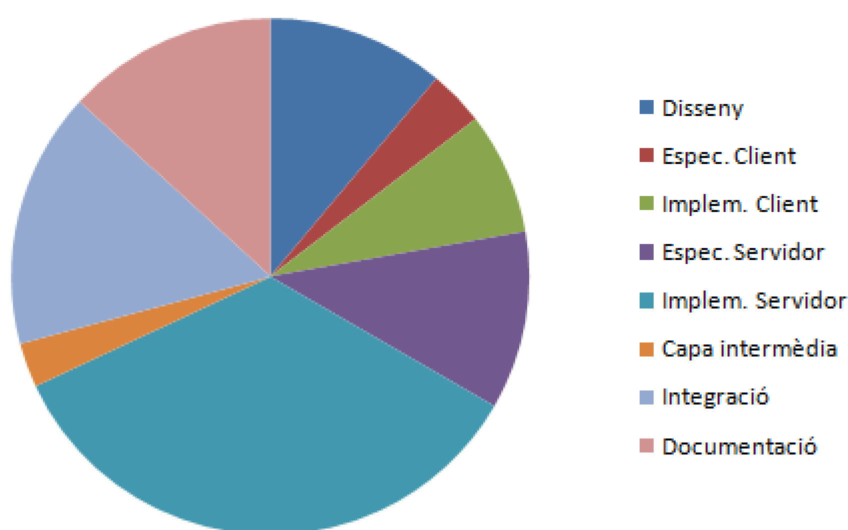
El diagrama de Gantt amb la planificació temporal de les vuit fases és el següent:



Es pot observar que hi ha un salt entre les fases d'implementació del client i especificació del servidor. Això es degut a que, per motius aliens a la planificació, es va haver de parar la feina de projecte durant un parell de setmanes, fins que vaig disposar de les eines per a poder realitzar la següent fase. Va tractar-se d'un imprevist que va endarrerir l'inici de la següent fase, i en part va propiciar la falta de temps que ha provocat que no tots els tests que s'havien plantejat inicialment s'hagin pogut dur a terme.

Tot i que el diagrama de Gantt insinua que la part de disseny és la que més temps ha consumit, Això no es compleix. El temps que he dedicat al projecte ha anat incrementant en el temps, i per tant el nombre d'hores invertides en les primeres fases no és el mateix que el nombre d'hores invertides en les darreres.

Aquesta gràfica mostra aproximadament el nombre d'hores que s'ha invertit en cada una de les vuit fases de desenvolupament del projecte.



Anàlisi de costos

El nombre d'hores que s'ha invertit per a cada fase és el que es mostra en la següent taula. Les dades són les que s'han utilitzat per a calcular la gràfica circular que hem vist a la secció anterior.

Fase	Durada (en hores)
Disseny	80
Especificació del client	25
Implementació del client	55
Especificació del servidor	80
Implementació del servidor	250
Capa intermèdia	20
Integració	115
Documentació	95
TOTAL	**730**

Per a calcular el cost en salaris del projecte, hem suposat dos perfils diferents. Per una banda, necessitem un cap de projecte que s'encarregui de coordinar les

diferents parts. A més, també s'encarregarà de dur a terme el rol de dissenyador i especificador. Per altra banda, donat que en el projecte el temps d'implementació té un pes important, comptarem amb dos programadors que s'encarreguin d'implementar els components client i servidor.

- Perfil: cap de projecte
  - **Salari**: 20 euros per hora.
  - **Seguretat social**: un 30% del salari
  - **Fases**: Disseny, Especificació del client, Especificació del servidor, part de la Integració, Documentació.
  - **Nombre total d'hores:** 295
- Perfil: programador
  - **Salari:** 12 euros per hora.
  - **Seguretat social**: un 30% del salari
  - **Fases:** Implementació del client, Implementació del servidor, Capa intermèdia, part de la Integració.
  - **Nombre total d'hores (entre els dos):** 435

A partir d'aquesta definició de rols, podem calcular el cost dels salaris de l'equip que treballarà en el projecte. Aquest cost serà també el cost total de l'aplicació, donat que no estem facilitant hardware de cap mena i que tot el software de tercers que utilitzem per al desenvolupament és de codi obert. Així doncs, el cost total de l'aplicació es pot calcular de la següent manera:

Cost total del projecte = 1.3 x 20 x 295 + 1.3 x 12 x 435 = 14.456 €

Conclusions

Una de les conclusions més clares que he pogut treure d'aquest projecte és que hi ha un motiu molt clar pel qual la tecnologia de *Cloud Computing* està en alça: l'estil de programació és molt diferent, i aplicacions que en un paradigma tradicional de client-servidor serien gairebé trivials d'implementar, es poden tornar molt complexes quan considerem la possibilitat de fraccionar-les en nuclis d'execució més petits per tal d'aprofitar millor els recursos disponibles.

A nivell menys tecnològic, i més de metodologia, aquest projecte m'ha servit per adonar-me d'una gran veritat: no convé subestimar el temps que portarà dur a terme una tasca. Com hem pogut veure en l'apartat de planificació, el temps que s'ha invertit en cada fase i el nombre de dies que ha durat cadascuna no coincideixen bé. L'últim mes i mig de la planificació ha sigut un mes de treball a contrarellotge, per culpa d'una mala apreciació a l'hora d'estimar el temps que duria implementar parts que estaven perfectament ben especificades.

La corba d'aprenentatge de la llibreria *CoDeS* ha resultat ser més llarga del que esperava. Es tracta d'una llibreria molt potent, però alhora molt extensa. Tot i la documentació existent sobre les metodologies i procediments per a efectuar les accions, ha calgut més temps del que inicialment estava previst per a efectuar la implementació de la part més elaborada del codi del projecte.

Tanmateix, a grans trets, la conclusió més important que puc treure després d'haver fet aquest treball és que iniciatives com *CoDeS*, o com aquest mateix projecte, no són tan idealistes com hom podria pensar. La demanda de recursos software és un problema real de la informàtica actual. I la oferta de recursos hardware desaprofitats, lluny de ser un problema, també és una característica important de la informàtica actual, com a mínim en la immensa majoria de casos. Que es desenvolupin infraestructures que s'encarreguin de relacionar aquests dos factors per tal de que s'aprofitin mútuament, és simplement una simbiosi perfecta. La tecnologia actual ho permet, només cal que es desenvolupi més software aprofitant aquesta tecnologia.

Valoració personal

Fa sis mesos que vaig inscriure aquest projecte. Han sigut sis mesos de treball dur, combinant una feina a mitja jornada, una assignatura optativa i l'elaboració d'aquest projecte. Dels sis mesos, concretament els dos últims han sigut un contrarellotge constant, en el qual la meva vida social ha desaparegut al complet. Tanmateix, crec que és la millor decisió que he pres en molt temps. Matricular un projecte per a fer-lo en un únic quadrimestre, amb la pressió de saber que si no arribava a temps per a acabar-lo hauria de tornar a matricular l'assignatura, és el millor que he fet en molts anys de carrera.

En les conclusions del treball he posat que la corba d'aprenentatge de *CoDeS* és molt llarga. Tot i que he comptat amb la inestimable ajuda d'un dels desenvolupadors de la llibreria, que m'ha ajudat molt sobretot en la part de preparació de l'entorn d'execució, he passat una gran quantitat d'hores llegint codi font, desembolicant la llibreria mica a mica, per entendre per quin motiu les coses funcionaven o deixaven de funcionar. Per a una persona com jo, que porta molts anys fent la carrera i que pràcticament ha perdut tot interès en feina dura d'aquest tipus, ha resultat estranyament refrescant.

L'única valoració negativa que puc fer respecte a aquest projecte, és que m'hagués agradat haver tingut temps per a implementar totes les funcionalitats que havíem pensat en el disseny. A part d'això, només se m'acudeixen valoracions personals positives. Cada una de les moltes hores que he invertit en aquest projecte, l'he invertit de bon grat. He aprés moltíssim i confio en que l'aportació que he realitzat al projecte *CoDeS* de la Universitat Oberta de Catalunya sigui d'utilitat per al desenvolupament d'aquesta eina que, en la meva humil opinió, presenta una manera de treballar innovadora que pot arribar a marcar tendéncia.

Referències

[1] http://www.facebook.com
[2] http://www.twitter.es
[3] http://es.wikipedia.org/wiki/Computacion_en_nube
[4] http://dpcs.uoc.edu/projects/codes/index.html
[5] http://es.wikipedia.org/wiki/Interfaz_grafica_de_usuario
[6] http://es.wikipedia.org/wiki/Localhost
[7] http://es.wikipedia.org/wiki/Balance_de_carga
[8] http://www.w3.org/html/
[9] http://www.subgurim.net/Articulos/ajax-y-javascript/54/ajax-a-pelo-xmlhttprequest.aspx
[10] http://es.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[11] http://www.java.com/es/
[12] http://es.wikipedia.org/wiki/Java_Servlet
[13] http://jetty.codehaus.org/jetty/
[14] http://es.wikipedia.org/wiki/Arquitectura_Orientada_a_Servicios
[15] http://en.wikipedia.org/wiki/String_(computer_science)
[16] http://en.wikipedia.org/wiki/Uniform_Resource_Locator
[17] http://en.wikipedia.org/wiki/Long_integer
[18] http://en.wikipedia.org/wiki/Epoch_(reference_date)
[19] http://support.twitter.com/articles/164083-what-is-a-timeline
[20] http://www.screaming-penguin.com/node/7570
[21] http://blog.danieldee.com/2009/06/callback-vs-listener.html
[22] http://es.wikipedia.org/wiki/Script
[23] http://es.wikipedia.org/wiki/Singleton
[24] http://en.wikipedia.org/wiki/Third-party_software_component
[25] http://en.wikipedia.org/wiki/Key-based_routing
[26] http://en.wikipedia.org/wiki/Distributed_hash_table
[27] http://en.wikipedia.org/wiki/ZIP_(file_format)
[28] http://download.oracle.com/javase/tutorial/networking/sockets/clientServer.html
[29] http://en.wikipedia.org/wiki/Pool_(computer_science)
[30] http://es.wikipedia.org/wiki/Java_Remote_Method_Invocation
[31] http://www.jguru.com/faq/view.jsp?EID=560072
[32] http://en.wikipedia.org/wiki/Timeout_(computing)
[33] http://en.wikipedia.org/wiki/Brainstorming

Bibliografia

- **Wikipedia, the free encyclopedia** (múltiples autors). http://es.wikipedia.com (última consulta, 20-06-2011).
- **CoDeS, A middleware for Contributory Communities** (Daniel Làzaro, entre altres). http://dpcs.uoc.edu/projects/codes/index.html (última consulta: 19-06-2011).
- **Java Programming API** (múltiples autors). http://download.oracle.com/javase/7/docs/api/ (última consulta: 12-06-2011)
- **Twitter: social networking and microblogging** (Twitter). http://www.twitter.es (última consulta: 20-06-2011)
- **World Wide Web Consortium (W3C)** (múltiples autors). http://www.w3c.org (última consulta: 01-06-2011
- **Pastry, a substrate for peer-to-peer applications** (Jeff Hoye) http://www.freepastry.org (última consulta: 15-05-2011).
- **Status.net - Enterprise Social Software is OPEN for Business** (Evan Prodromou) http://status.net (última consulta: 12-02-2011).

Annex: Instruccions de desplegament de HorNet

Durant tota la documentació hem pogut observar el procés de desenvolupament de HorNet en totes les seves fases. Tanmateix, juntament amb aquesta documentació també es facilita un arxiu comprimit, ***HorNet.tar.gz***, que conté tot el codi del projecte. En aquest annex veurem com està estrcturat aquest arxiu i quines són les instruccions de desplegament per a l'aplicació.

HorNet.tar.gz

Aquest arxiu conté una carpeta anomenada HorNet, que representarà l'arrel de la jerarquia de l'entorn d'execució. Aquesta arrel conté el següent:

- **bin**: directori contenint els arxius binaris (.class) de HorNet
- **src**: directori contenint els arxius de codi (.java) de HorNet
- **lib**: directori contenint les llibreries (.jar) necessàries per a l'execució de HorNet
- **test**: directori contenint els arxius de codi (.java) utilitzats per al testing dels serveis de HorNet
- **Srv*.zip**: son viut fitxers contenint els serveis que es desplegaran a CoDeS per al correcte funcionament de HorNet.
- **configAPI.properties**: arxiu de configuració de CoDeS, necessari per a que HorNet sàpiga on localitzar l'API per a realitzar la comunicació amb la xarxa.
- **style.css**: arxiu d'estil, necessari per al correcte format del client web
- **ProfileTemplate.xml**: arxiu de configuració de HorNet, necessari per a establir quins paràmetres volem que formin part del perfil d'usuari.
- **template.raw**: arxiu de text pla que conté, simplement, la paraula "new". Aquest arxiu és necessari per al correcte desplegament dels serveis.

Precondicions d'execució

HorNet té dos components executables, un corresponent al contenidor web i l'altre corresponent al servidor (el client, recordem, és un navegador web qualsevol). Ambdós components tenen un requeriment important, i és que la màquina on s'estiguin executant estigui formant part d'una xarxa de CoDeS. El desplegament d'aquesta xarxa comunitària no entra dins l'abast d'aquest projecte. Per a més informació sobre com desplegar una xarxa de CoDeS o sobre com incorporar-se a una ja existent, cal consultar la web dels autors: dpcs.uoc.edu/projects/codes/index.html

Per descomptat, al tractar-se d'una aplicació java, caldrà que una màquina virtual de Java estigui instal·lada i executant-se en el node on vulguem executar HorNet.

A més, cal instal·lar Jetty, un programa de codi lliure multi plataforma. Es pot descarregar de la pàgina dels seus autors: http://jetty.codehaus.org/jetty/

Execució dels components

Servidor

El servidor es pot executar mitjançant la següent comanda, introduïda des del context del directori HorNet/ :

```
# java -cp HorNet.jar:lib/* LaunchHorNet
```

Si l'entorn està ben definit, tornarà un missatge similar a aquest:

```
# Authentication service successfully deployed
# Database file does not exist. Creating!
# Indexation service successfully deployed
# Database file does not exist. Creating!
```

I amb això ja disposarem dels serveis mínims per a executar HorNet a la xarxa.

Contenidor

Per a executar el contenidor, n'hi ha prou amb executar la següent comanda des del context del directori HorNet/:

```
# java -cp HorNet.jar:lib/* JettyLauncher
```

I ens retornarà una sortida similar a aquesta:

```
# EVENT Starting Jetty/4.2.15rc0
# EVENT Started ServletHttpContext[/]
# EVENT Started SocketListener on 0.0.0.0:8070
# EVENT Started org.mortbay.jetty.Server@158b649
```

I ja podrem executar el client, introduint l'adreça http://localhost:8070 en un navegador web.