ideaD Aplicació d'intercanvi de documents en entorns asíncrons i distribuïts

Projecte:	**ideaD Aplicació per a l'intercanvi de documents en entorns asíncrons i distribuïts**
Tipus:	**Treball fi de carrera**
Document:	**Memòria**
Versió:	**0001**
Carrera:	**Enginyeria Tècnica en Informàtica de Gestió**
Àrea:	**Aplicacions i sistemes distribuïts**
Estat:	**Definitiu**
Data:	**10/12/05**
Autor:	**Amadeu Marsà i Sandiumenge**
	mail to:amarsa@uoc.edu
Consultor:	**Carles Pairot i Gavaldà**
Universitat:	***Universitat Oberta de Catalunya - UOC***
	Internet: **http://www.uoc.edu**

Índex

Introducció

Presentació del projecte

El projecte ideaD és un entorn per a l'intercanvi de documents i informació en general per a grups de treball asíncrons i distribuïts. Entenem per asíncrons aquells grups els membres dels quals no es troben en un mateix moment del temps i distribuïts com a aquells en què la informació no es troba en un mateix lloc.

La necessitat de l'intercanvi d'informació en aquest tipus de grups és bàsica, havent de garantir que cada usuari tingui en tot moment una versió actualitzada dels documents de treball. Per fer-ho ideaD agafa els serveis de LaCOLLA de distribució d'informació i events i d'emmagatzematge d'informació de forma replicada i descentralitzada.

La tecnologia emprada per la solució és d'aplicació web, per la seva portabilitat i facilitat de distribució.

Objectius

Objectius del treball fi de carrera:

- Aprendre a fer aplicacions usant servlets o JSP;
- Utilitzar XML;
- Conèixer les necessitats de les aplicacions col·laboratives;
- Adoptar solucions per aquesta problemàtica;
- Conèixer la problemàtica de les aplicacions d'igual a igual (peer – to – peer), descentralitzades i distribuïdes.

Definició del projecte

Pla de treball

El pla de treball inicial es va pensar de la següent forma:

Funcionalitats a implementar

S'ha decidit començar amb alguna tasca relativament senzilla per a familiaritzar-se progressivament amb el nou entorn: les idees del programari peer to peer, l'api de LaCOLLA, els servlets i pàgines JSP, etc... per anar incrementant la complexitat de les funcionalitats implementades. L'objectiu del projecte és acabar amb un nucli de serveis oferts per l'aplicació que compleixi els requeriments i pugui anar sent millorat de forma escalable. S'ha decidit:

El **primer pas** consisteix en decidir una **estructura** prou flexible i completa on desar les dades del directori virtual. Per a això s'ha decidit utilitzar la tecnologia xml i, en aquest sentit, es definirà una estructura de nodes i atributs per desar la informació necessària per al treball en local.

En **segon lloc** es triarà alguna de les **tecnologies** disponibles **per l'accés a fitxers xml** en Java (JDOM, dom4j, SAX, DOM) i es desenvoluparan les classes necessàries per assolir certes funcionalitats com la navegació per l'estructura de fitxers, l'alta de nova informació, l'eliminació (si és possible en una estructura replicada), etc... ja pensant en la seva implementació posterior en pàgines JSP i servlets.

De forma simultània en aquest punt 2, a mesura que es completin certs conjunts de funcionalitats, es passarà a generar la seva visualització per navegador mitjançant les pàgines JSP.

Igualment de forma simultània, s'integrarà la interfície de LaCOLLA.

Finalment, es realitzaran les proves i modificacions necessàries per al funcionament en grup: la tasca se centrarà en els esdeveniments i la propagació de la informació entre els iguals.

Tasques del projecte

1. Anàlisi prèvia, que inclou:

   - Anàlisi de programes similars per determinar les tasques a implementar (BSCW);
   - Conèixer el funcionament de LaCOLLA (API, etc...);
   - Familiarització amb les idees del treball peer to peer (P2P);
   - Millorar els coneixements en llenguatge de programació Java de servlets / JSP i RMI, a més de la base del llenguatge XML i el seu enllaç amb Java (JDOM);

2. Anàlisi i disseny de les estructures de dades necessàries (xml)
3. Anàlisi i disseny de les classes – Beans necessaris per a dur a terme les funcionalitats parlades de navegació per l'estructura de carpetes, alta d'objectes, modificació i eliminació. Inclou la realització dels tests individuals d'aquestes classes.
4. Creació de la classe (o classes) de comunicació amb LaCOLLA: implementació de la interfície remota per la captura d'esdeveniments i el seu tractament. Inclou la realització dels tests individuals d'aquestes classes.
5. Creació de l'entorn visual en pàgines JSP i proves del mateix;
6. Integració amb LaCOLLA i proves en "mono-lloc";
7. Proves del sistema global en simulador i, després, en un entorn real. Inclou aprendre a usar el simulador de xarxa SIM o bé PlanetLab;
8. Memòria i comparativa amb altres solucions al mercat, que inclou:

   - Compilació de la documentació generada;

- Cercar i conèixer d'altres solucions possibles en aquesta àrea com BSCW o Groove (comparativa);

Calendari

Tasca	Durada	Inici	Fi	Prerequisits
1. Anàlisi prèvia: Anàlisi de programes similars; Estudi de l'API de LaCOLLA; Estudi de Servlets, pàgines JSP, RMI, XML,...	12 dies	20/09/05	09/10/05	
2. Estructura de dades: Definició de l'estructura de dades; Generació d'un fitxer xml d'exemple;	3 dies	10/10/05	15/10/05	1
3. Classes base: Classes abstracció dels objectes; Lectura / serialització en xml; Tests;	11 dies	15/10/05	04/11/05	1, 2
4. Fase RMI: Connexió RMI amb LaCOLLA;	11 dies	04/11/05	21/11/05	1, 2, 3

Implementació parcial API client; Test;				
5. Fase web: Formularis; Finestres de treball; Test;	6 dies	21/11/05	28/11/05	1, 2, 3
6. Integració LaCOLLA: Implementació api client; Test	15 dies	03/12/05	27/12/05	4, 5
7. Documentació: Memòria; Presentació;	8 dies	27/12/05	09/01/06	6

Figura 1 - Calendari tasques del projecte

Com a resultat deixa el següent diagrama de Gant:

Diagrama de Gant

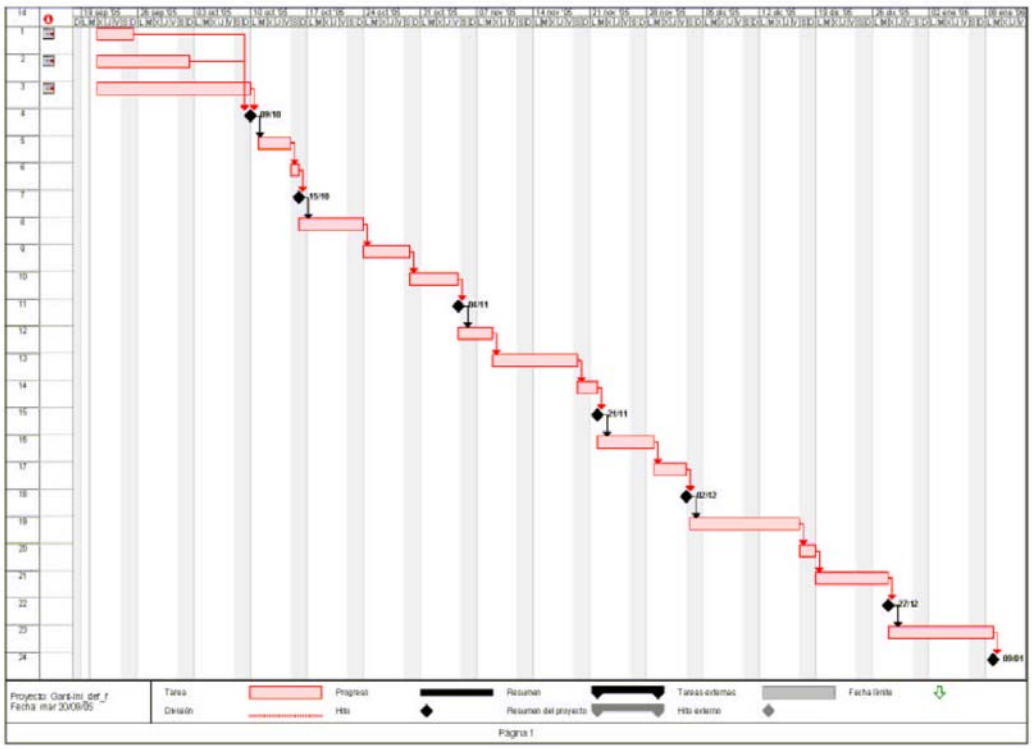

Figura 2 - Diagrama de Gant inicial del projecte

Fita 1 – Anàlisi prèvia

Els primers passos del projecte van delimitar l'abast d'aquest: en primer lloc calia pensar quines necessitats representava un entorn col·laboratiu d'intercanvi de fitxers com ideaD. Per fer-ho calia saber quines funcionalitats s'havien implementar.

Anàlisi d'entorns col·laboratius

La majoria d'entorns de treball col·laboratius són centralitzats, degut a les facilitats de gestió que aporta una arquitectura centralitzada. Tot i aquest fet, es

poden utilitzar per determinar les accions que els usuaris voldran fer sobre els objectes.

Els espais compartits en servidors i sota sistemes d'arxius Windows/Unix

Segurament són els entorns col·laboratius més usats al món: els propis sistemes operatius ofereixen molts serveis útils per al treball en equip ja que disposen de:

- Una forma de classificar la documentació en carpetes estructurable al gust de l'usuari;
- Sistema de privilegis sobre els objectes continguts que permeten determinar quines accions pot realitzar cada usuari sobre cada objecte (en concret, per al projecte, s'havia dissenyat una còpia del que fa servir Linux/Unix, amb un sistema de nou caràcters, agrupats de tres en tres, per a l'usuari, el grup i la resta);
- Possibilitats de manipulació de fitxers: desat de nous fitxers, substitució de fitxers, moviment de fitxers en l'espai de carpetes, ordenat de la visualització, recerques, etc...

El *Basic Support for Collaborative Work* (BSCW)

Eina molt més completa i interessant de cara al projecte de ideaD, donat que està pensada precisament per cobrir les necessitats dels grups de treball asíncrons i per ser utilitzada per Internet. Això sí: segueix sense ser distribuïda ni descentralitzada, donat que desa còpies de la tota la informació de forma centralitzada perquè estigui permanentment disponible.

Com els entorns de fitxers, manté els objectes en estructures arbòries de carpetes, encara que més aviat hauríem de parlar de *contenidors*, ja que té estructures especialitzades com debats, encarregats només de desar missatges.

Per la coordinació dels membres incorpora un sistema d'esdeveniments que permet realitzar accions sobre objectes (donar per vist o llegit) i el motor del BSCW els reenvia als participants del mateix grup. En aquest sentit seria molt interessant agregar a ideaD els serveis de missatgeria instantània d'alguna producte.

Lotus Domino

Producte també força complet que mira de ser un entorn total per al treball en grup. Evidentment també és centralitzat (servidor) i disposa d'un conjunt de funcionalitats molt completes per al treball en equip, encara que potser l'entorn per a l'intercanvi de documents és una mica més fluix que el del BSCW.

Arquitectura

Seguidament analitzarem dues possibles arquitectures de treball amb ideaD, depenent del concepte que vulguem aplicar a la idea d'*igual* en aplicacions *peer – to – peer*.

P2P per ordinador

Arquitectura en què es defineix com a igual cada usuari de ideaD. En aquest cas és necessari que hi hagi una instància de LaColla corrent en cada ordinador, juntament amb el servidor web per servir l'aplicació.

Els iguals comparteixen els recursos d'una forma totalment descentralitzada, essent LaColla qui gestiona els recursos per evitar fallades en cas de desconnexió.

P2P per organització

Els iguals en aquesta relació són les organitzacions que comparteixen informació, representades en els seus conjunts de servidors de LaColla que interaccionen per la xarxa.

En primer lloc necessitem un servidor d'aplicacions web amb extensió per a interpretar Servlets i pàgines JSP per a ideaD i, a més, necessitarem un servidor per a LaCOLLA. Aquests dos es comunicaran via *Remote Method Invocation (RMI)* i, per tant, necessitarem tenir instal·lada la màquina virtual de Java.

Els ordinadors clients no necessitaran de cap instal·lació, ja que seran clients lleugers que amb un simple navegador podran tenir accés a ideaD. Així, segons es necessiti, l'aplicació estarà disponible tant en la intranet de l'organització com, si es vol, externament.

A partir d'aquí, el nostre servidor de LaCOLLA podrà posar-se en contacte via internet amb d'altres servidors de LaCOLLA del mateix grup per a l'intercanvi d'informació.

Aquesta arquitectura presenta com a avantatge (propi de tota arquitectura client/servidor per aplicacions web) de ser fàcilment extensible a nous clients (escalabilitat), ja que aquests no necessiten d'instal·lació: l'únic coll d'ampolla a tractar en cas d'ampliacions és la capacitat del/s servidor/s d'aplicacions web i LaColla, així com els recursos que es posen a disposició d'aquesta (emmagatzematge, execució de tasques,...).

A més, si és vol evitar fallades dins l'organització és pot pensar en descentralitzar la informació amb diverses instàncies de LaColla i tenir un programari intermedi que actuï com a balancejador de càrrega i distribueixi les peticions entre els servidors. D'aquesta forma, en cas de caiguda d'un dels servidors es podrà seguir treballant.

Un altre dels avantatges és la definició dels usuaris: l'organització pot tenir un base de dades corporativa a la qual es podria connectar ideaD per obtenir-ne els usuaris. D'aquesta forma també es poden definir rols (a ideaD n'hi ha dos: administrador i usuari) per configurar l'aplicació segons les necessitats de l'usuari.

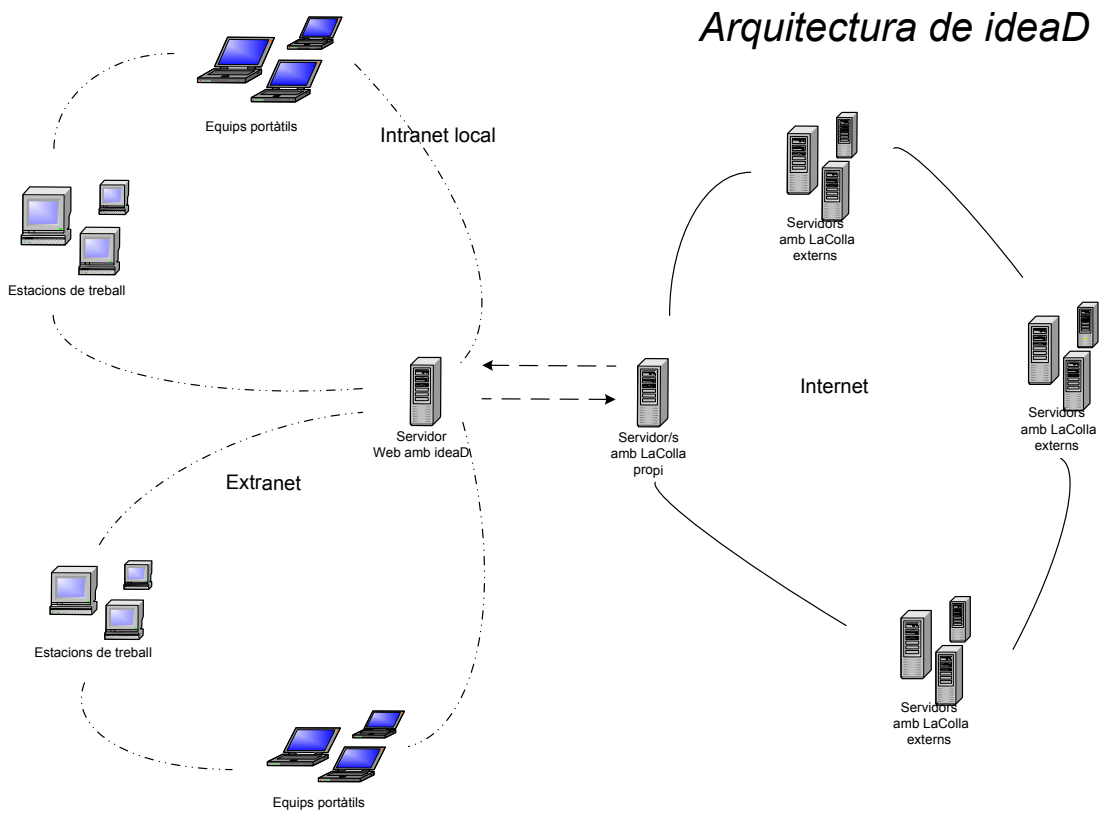

Figura 3 – Arquitectura de ideaD per organitzacions

Tria de la tecnologia

JDOM per tractar XML

Com es veu pel títol de la secció, s'ha triat com a tecnologia de programació per a tractar els documents XML JDOM.

JDOM és una API totalment en Java i de codi lliure per a transformar, crear, manipular i serialitzar documents XML. D'altres tecnologies que podrien haver servit són DOM, SAX o dom4j.

Més o menys totes aquestes tecnologies proveïen les funcionalitats que es necessitaven, però donades les característiques del projecte i la necessitat d'un ràpid aprenentatge, JDOM semblava la que més intuïtivament s'adaptava a la feina a fer.

Com DOM, JDOM representa un document XML com un arbre compost d'elements, atributs, comentaris, instruccions, CDATA,... Tot l'arbre sencer està disponible en tot moment, a diferència de amb SAX, a més de disposar d'unes interfícies a implementar per fer filtres força pràctiques. Com a avantatge fonamental en quant a no perdre temps en aprenentatge és que a diferència de DOM i 4domj cada tipus de node es representat per classes concretes i no per interfícies.

També cal destacar la intuïtivitat de la forma de crear i treballar en general amb els nodes, així com la facilitat per serialitzar després en fitxer la feina feta, amb classes ja preparades per deixar el fitxer visible, és a dir, fixar els salts de línia o les sangries segons el nivell d'una forma automatitzada.

Pàgines JSP

Avantatges de les pàgines JSP sobre els Servlets

La tecnologia Java Servlet és la base de totes les tecnologies de Java per al desenvolupament d'aplicacions web, fins al punt que es recomana familiaritzar-se amb aquesta tecnologia fins i tot en el cas de no voler usar-la. De fet la tecnologia Java Server Pages no és més que una extensió definida al damunt de l'API dels Servlets, és a dir, una capa que oculta certes complexitats dels servlets al programador, ja que tota pàgina JSP és transformada pel contenidor web en un servlet i compilada com a tal.

Per tant, en la seva mateixa definició ja hi tenim un dels avantatges: es van crear precisament per ocultar (el compilador genera codi automàticament) certes complexitats dels servlets, sobre tot en referència a visualització.

Cicle de vida d'una pàgina JSP

Quan una petició va dirigida a una pàgina JSP, primer, el contenidor web verifica si el servlet associat és més antic que la pàgina. Si el servlet és obsolet, aleshores torna a transformar la pàgina JSP en un servlet per compilar aquest i passar-lo al contenidor de servlets on, des d'aquest moment, l'objecte encarregat de respondre a les peticions d'aquella url seguirà el cicle de vida clàssic d'un servlet, amb els mètodes init() i destroy() per iniciar-lo i destruir-lo, respectivament i el service() per tractar les peticions i generar les respostes.

És un altre avantatge fonamental per al programador de les pàgines jsp sobre el servlets: el *dinamic reloading*. Quan fem qualsevol canvi sobre el fitxer font (i

desem), aquest és detectat pel servidor que refà el servlet derivat i recompila; mentre que amb servlets hem de forçar la compilació nosaltres mateixos.

Una tecnologia orientada a la presentació

El principal problema que presenten els servlets és la dificultat de generar el contingut estàtic (presentació) de la pàgina web. Al ser una tecnologia orientada a la programació (per programació volem dir lògica de l'aplicació) no es va pensar en facilitar la feina de la presentació, amb què el codi destinat a aquesta part ha d'anar imbuït en el teixit del codi encarregat de la lògica de negoci. Per contra, les pàgines JSP ja van ser ideades per resoldre aquest problema, no només facilitant un conjunt de tags per ajudar a crear aquest contingut estàtic, sinó que per la seva concepció és una tecnologia ideal per poder generar codi automàticament des d'aplicacions de programació WYSIWYG. A més a més de poder estendre totes aquestes funcionalitats amb llibreries de tags pròpies o, la última de les tecnologies, els Java Server Faces.

Aquesta raó ja va ser suficient per optar per generar totes les pàgines amb contingut estàtic del projecte (tota la presentació) en forma de pàgines JSP o bé amb una combinació de pàgines JSP encarregades del contingut estàtic que incloïen servlets per la part dinàmica. En el primer cas, trobem els molts formularis del projecte, mentre el segon és el cas de la pàgina principal de treball, la part estàtica de la qual és una pàgina JSP mentre que la que s'encarrega de llistar el contingut del directori de treball és un servlet.

Per què també pàgines JSP de control?

Si mirem els *'Blue papers de Java'* les pàgines JSP semblen enclaustrades a jugar un simple paper de presentació. Això és així perquè van ser ideades per servir aquest objectiu en el qual els servlets, com ja s'ha comentat, no havien estat gens reeixits. Així doncs, sota el principi de separació de lògica de negoci i presentació, l'associació d'idees sembla clara: la tecnologia pensada per la presentació (pàgines JSP) només s'ocuparà d'aquest tema i els servlets carregaran amb el negoci.

Ara bé, si reflexionem una pàgina JSP no és més que una abstracció d'alt nivell d'un servlet i, per tant, pot fer qualsevol cosa que faci un servlet, incloent encarregar-se de la lògica de negoci de la nostra aplicació. En segon lloc, no cal caure en la simplificació de servlets per negoci i pàgines JSP per presentació com a paradigma de la bona programació: el principi només diu separació de lògica i presentació. Per tant, el seguirem complint si hi ha pàgines JSP només encarregades de la presentació i pàgines JSP que no fan res de presentació i només s'ocupen de la lògica de negoci.

Anàlisi: altres. La problemàtica del treball *off-line*

Un dels reptes més importants que planteja el projecte és poder oferir els mateixos (o, si més no, gairebé els mateixos) serveis als usuaris en cas que estiguin treballant desconnectats, és a dir, la seva instància de LaCOLLA estigui fora de línia amb les altres instàncies de LaCOLLA del grup.

Part dels problemes d'aquest tipus de treball vénen resolts per les funcionalitats de disseminació d'esdeveniments de LaCOLLA, que haurtien de garantir que en la reconnexió totes les accions realitzades fossin rebudes per cada membre del grup: l'usuari desconnectat rebria els esdeveniments que s'hauria perdut generats en el grup i, a la inversa, els usuaris del grup rebrien els esdeveniments de l'usuari que havia estat fora de línia.

Però què passa amb esdeveniments contradictoris? Un dels casos més clars i sagnants és el de l'eliminació: si el grup ha eliminat una carpeta i l'usuari desconnectat hi ha pujat un nou objecte, com han de tractar cada part cada un dels esdeveniments?

Es va estimar que seria necessari l'elaboració de tot un procediment de reconnexió segur, en el qual l'usuari que es reconnecta hauria de fer un paper de servidor i rebre el fitxer del directori virtual del grup, comparar-lo amb el propi, establir si totes les accions eren possibles i, en cas de fallada, prendre decisions.

Davant dels enormes costos en implementació d'aquestes polítiques, es va decidir deixar fora de la'bast d'aquest projecte el tema del treball *off-line*.

Fita 2 - Estructura i emmagatzematge de dades

Per la realització del projecte es necessita emmagatzemar en el servidor de ideaD l'estructura virtual de carpetes de treball del directori sobre la qual l'usuari ha de tenir la sensació de treballar, ja que LaCOLLA només s'encarrega de mantenir fitxers reals. A més, també es fa necessari mantenir dos tipus de dades: d'una banda les derivades de la configuració del programari i, de l'altra, les provinents de la gestió d'usuaris pròpia de l'arquitectura proposada per a ideaD com a peer – to – peer per organització.

Configuració

El fitxer de configuració és una simple cistella de recursos de text pla, amb una clau i una entrada per clau. S'ha optat per aquesta solució donada la poca quantitat de dades i la seva baixa mutabilitat.

Estructura del directori virtual

Pel directori virtual hom s'ha decidit per implementar-la via fitxer xml. En primer lloc perquè l'aprendre a tractar aquesta tecnologia era un dels objectius del treball fi de carrera, però també perquè la quantitat de dades a emmagatzemar no era tant gran com per ser necessaris els serveis d'una base de dades relacional per mantenir-los. Això sí, en una organització amb un ús intensiu i molts canvis en l'estructura del directori podria notar-se una caiguda en el rendiment de l'aplicació deguda a la necessitat d'anar refent el fitxer una vegada i una altra. Per tant, en aquests casos seria necessari crear una versió en base de dades.

Una altra de les raons per triar un fitxer xml és la capacitat de convertir-lo en un fitxer més de LaCOLLA, de tal forma que si un nou servidor es vol connectar a treballar només caldria que demanés el fitxer (mentre que amb base de dades, hauria de restaurar còpia i ens trobaríem amb el problema què entre la data de la còpia i la restauració, si l'equip ha seguit treballant, l'estructura hagi sofert canvis; és a dir, segurament implicaria una aturada del treball).

En el fitxer hi trobarem:

- ✓ Un únic element *root*, element arrel del fitxer i que podrà contenir elements de tipus carpeta, fitxer i nota;

L'estructura és:

Element root	
Atributs	
▪ idObject	Identificador i gual a *nul* per defecte.
Podrà contenir elements	
➢ carpeta	
➢ fitxer	
➢ nota	

Figura 4 - Estructura de l'element root

- ✓ Elements que representaran els objectes del nostre programari: *carpeta*, *fitxer* i *nota*;

La seva estructura és:

Element carpeta	
Atributs	
▪ idObject	Identificador.
▪ nom	Nom de l'objecte a visualitzar
▪ data	Data de creació
▪ propietari	Usuari que va crear la carpeta
▪ grup	Grup de l'usuari que va crear la carpeta
▪ drets	Cadena de tres grups de tres lletres representant els drets de l'usuari, el grup i el món sobre l'objecte en forma de R – lectura, W – escriptura i X – execució.
▪ obs	Cadena de caràcters amb informació sobre l'objecte.
Podrà contenir elements	
➢ carpeta	
➢ fitxer	
➢ nota	

Figura 5 - Estructura d'un element carpeta

Element fitxer	
Atributs	
▪ idObject	Identificador
▪ nom	Nom de l'objecte a visualitzar
▪ tipus	Extensió del fitxer
▪ mida	Mida en bytes
▪ data	Data de creació
▪ propietari	Usuari que va crear la carpeta
▪ grup	Grup de l'usuari que va crear la carpeta
▪ drets	Cadena de tres grups de tres lletres representant els drets de l'usuari, el grup i el món sobre l'objecte en forma de R – lectura, W – escriptura i X – execució.
▪ obs	Cadena de caràcters amb informació sobre l'objecte.
▪ versió	Versió del fitxer
▪ valoració	Indicació de com es valora l'objecte
Podrà contenir elements	
➢ vist	
➢ llegit	
➢ enUs	

Figura 6 - Estructura d'un element fitxer

Element nota	
Atributs	
▪ idObject	Identificador
▪ tema	Tema del missatge
▪ respon	Tema del missatge al qual respon
▪ data	Data de creació
▪ propietari	Usuari que va crear la carpeta
▪ grup	Grup de l'usuari que va crear la carpeta
▪ drets	Cadena de tres grups de tres lletres representant els drets de l'usuari, el grup i el món sobre l'objecte en forma de R – lectura, W – escriptura i X – execució.
▪ obs	Cadena de caràcters amb informació sobre l'objecte.
Contindrà l'element	
➢ text	Encapsula un element CDATA amb el missatge de la nota.
Podrà contenir elements	
➢ vist	
➢ llegit	
➢ enUs	

Figura 7 - Estructura d'un element nota

- ✓ Altres elements auxiliars per ajudar a representar la informació, com un element CDATA *text* per desar la informació de les notes; elements *vist* i *llegit* per informar quan un objecte ha estat llegit o vist i un element *enUs* per a implementar el sistema de bloquejos;

Element Vist / Llegit	
Atributs	
▪ qui	Usuari que ha realitzat l'acció
▪ quan	Moment en què s'ha realitzat

Figura 8 - Estructura d'un element vist o llegit

Molts dels camps estan pensats per facilitar la futura ampliació de les funcionalitats, en concret, de la gestió d'usuaris d'una banda, i la versionificació de fitxers per l'altra. Així l'atribut drets o les marques opcionals a afegir de enUs permetran garantir l'accés segur per grups als objectes i l'atribut versió dels fitxers permetrà mantenir diverses versions d'un element.

L'element enUs

Element enUS	
Atributs	
▪ valor	S ò X segons l'accés
▪ qui	Usuari que realitza l'accés

Figura 9 - Estructura d'un element enUs

La idea d'incorporar aquest element al programari era possibilitar un sistema de bloquejos que garantís l'accés concurrent als recursos de forma segura. Així els valors per aquest tipus són:

- S -> accés compartit (Shared) i
- X -> accés exclusiu (eXclusive);

Cada cop que un usuari volgués realitzar una acció hauria primer d'obtenir el bloqueig sobre l'objecte / els objectes triats corresponent al tipus d'esdeveniment. Així, per exemple, per descarregar-se un fitxer s'hauria d'assolir un accés compartit (S) i per eliminar-lo un accés exclusiu (X).

Evidentment, si un objecte té accessos compartits no es deixarà efectuar cap bloqueig exclusiu i, si un objecte té un marca d'accés exclusiu, no es deixarà fer cap bloqueig compartit o exclusiu, avortant-se l'acció.

L'objectiu d'aquesta arquitectura de bloquejos era possibilitar algunes accions tan bàsiques com eliminar objectes o retallar objectes sense les quals no són possibles.

A més, es podria generar un programari que gestionés cues de peticions de recursos, molt a l'estil d'una base de dades relacional.

Aquesta part ha quedat pendent d'implementar i seria una de les futures millores cabdals a realitzar.

Quedaria pendent d'analitzar com resoldre el problema d'eliminar els bloquejos quan un usuari tanca la sessió sense fer logout (i, per tant, deixa marques no existents).

Els atributs *propietari*, *grup* i *drets*

Pensats a imitació de la gestió de fitxers en Unix, aquests ens permeten determinar quines accions pot realitzar l'usuari actual sobre un objecte.

Com ja s'ha comentat, l'atribut drets emmagatzema una cadena de nou caràcters, agrupats de tres en tres. El primer grup de tres caràcters representa els drets del propietari, el segon els dels membres del mateix grup que el propietari i el tercer els de la resta d'usuaris.

Els valors que poden prendre són RWX o bé – si no està permesa el tipus d'acció. R implica permisos de lectura (*Read*), W permisos d'escriptura (*Write*) i X permisos d'execució (*eXecute*). Així podem trobar combinacions com R—(només lectura), R-X (lectura i execució però no modificació) i d'altres.

Cada acció que volgués un usuari dur a terme sobre un objecte implicaria la necessitat de tenir algun dels accessos anteriors actiu en el grup correcte.

És una altra de les millores que es proposen.

Gestió d'usuaris

En un principi estava completament aparcada del projecte inicial, però donades certes complicacions que van trigar força en solucionar-se es va decidir tirar endavant la creació d'una petita gestió d'usuaris pròpia de ideaD. Així es va decidir separar els usuaris de LaCOLLA dels propis de ideaD, copiant l'estructura de treball normal de moltes aplicacions distribuïdes de tipus client/servidor on l'aplicació que es comunica amb el gestor de bases de dades ho fa amb un usuari d'aquesta, mentre que els diversos clients es connecten a l'aplicació amb

usuaris propis d'aquesta. En la nostra idea del programari només existirà un usuari a LaCOLLA (encara que poden ser també tants com vulguin) per tota l'organització, mentre que la gestió d'usuaris de l'aplicació web serà pròpia, podent així, per exemple, configurar en el futur un sistema que permeti la captura dels usuaris d'una base de dades corporativa.

De cara a l'exterior (la comunicació amb d'altres servidors de LaCOLLA per compartir informació via Internet) evidentment no es podrien aprofitar serveis de missatgeria instantània o events per usuaris particulars de forma directa perquè aquests sempre arribaren dirigits a l'usuari corporatiu de LaCOLLA, però pactant que en l'estructura d'aquests events/missatges hi hagués informat l'usuari individual que els ha de rebre, es podria programar el codi necessari perquè fos només aquest qui els rebés. El sistema seria similar al de les xarxes actuals: tots els clients reben el "paquet", en aquest cas event, però només l'accepta (tracta) aquell que és l'usuari destinatari (l'aplicació miraria l'event, buscaria l'usuari entre una llista de sessions actives i només l'afegiria a la sessió pertinent).

Implementació

Eines utilitzades

El programari s'ha desenvolupat amb Eclipse 3.0.1, millorat amb el plug-in free-ware de Sysdeo per Tomcat, que permet treballar en projectes Tomcat sense necessitats de desplegament del projecte ni de canviar la ubicació dels fonts. S'ha utilitzat Tomcat 5.5.7 i el Java 1.5.0_01, per posteriorment migrar cap a Tomcat 5.5.12 i Java 5 Update 6.

Estructura dels paquets

El projecte s'ha generat amb la següent estructura de paquets:

a. "paquets web": carpetes amb què es divideix el directori del projecte creat per a les pàgines jsp;

   - *arrel*: inclou les pàgines jsp per loginar a l'aplicació;
   - *administrar*: conjunt de pàgines jsp per l'administració d'usuaris i grups de ideaD: només pàgines front-end, és a dir, encarregades de mostrar contingut;
   - *aplicacio*: pàgines jsp de comunicació amb l'usuari per tota la gestió de fitxers, carpetes i notes;
   - *proces*: totes les pàgines jsp de control, és a dir, no mostren cap informació a l'usuari, sinó que validen dades d'entrada o realitzen processos com el repartiment de peticions al component gestor encarregat de resoldre-la;
   - *util*: formularis jsp amb codi reaprofitable;

b. paquets "clàssics":

   - *accions*: paquet per a servlets encarregats de resoldre la gestió de les peticions. Per accions s'entén aquelles que pot realitzar l'usuari, com donar d'alta un nou objecte, sortir del programa, demanar un fitxer, etc...
   - *classes*: conté l'abstracció en codi Java dels objectes que tracta el programari. Totes les classes aquí definides, en ser necessària la

seva serialització i intercanvi via RMI, implementen la interfície *Serializable*.

UML estàtic del paquet Classes

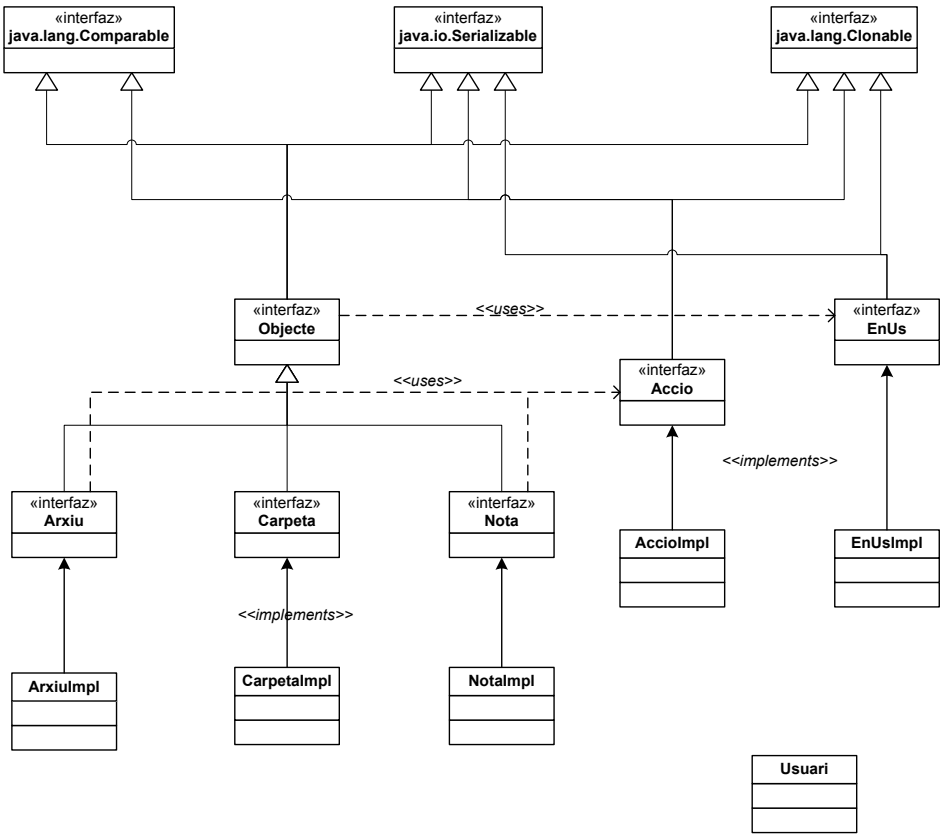

Figura 10 - UML estàtic del paquet Classes

- *directori.Api*: empaqueta les classes encarregades d'establir i gestionar la connexió amb l'interfície remota de LaCOLLA (Aplicacio) i de proporcionar la interfície de ideaD per les respostes / missatges de LaCOLLA (ApplicationsSideApiImpl).

- *events*: hi ha les classes necessàries per gestionar els events seguint el patró de disseny de factoria (per si es canvia l'api). Són les classes encarregades de crear i enviar els events del programari;

- *init*: gestiona la connexió inicial;

- *login*: gestiona la validació dels usuaris de ideaD;

- *mostra*: empaqueta servlets dedicats a tractar informació només per ser mostrada en pàgines jsp;

- *persistencia*: classes encarregades de gestionar la persistència en forma del patró de factoria per si es vol canviar aquest;

- *util*: diverses classes de suport en general;

- *xLaColla*: classes de suport per tractar amb els objectes de LaCOLLA;

- *xLaColla.event*: estructura pròpia d'events per ideaD;

- *xml*: classes encarregades de la serialització i lectura dels orígens xml amb què treballa ideaD.

Arquitectura de tres capes

S'ha seguit el suggeriment clàssic de separar interfície d'implementació. Així, les relacions entre els paquets mostren la separació entre paquets "de més baix nivell" dedicats a gestionar la persistència, és a dir, ocupar-se de les tasques de la representació física dels objectes, els paquets que gestionen el model i els que es dediquen a la presentació.

En la següent figura veiem com els paquets *events* i *persistencia* són els únics que realitzen accessos als paquets *xml*, dedicat a la persistència d'objectes en fitxers xml i *xLaColla*, dedicat a generar els objectes esdeveniment per a LaCOLLA.

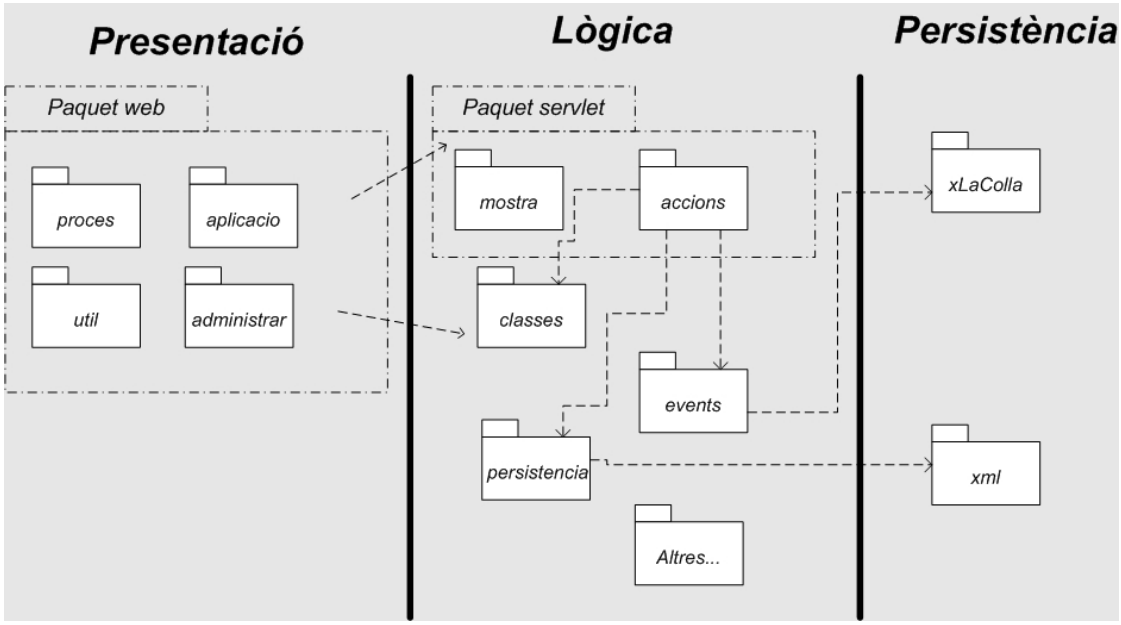

Figura 11 – Arquitectura de tres capes

Al seu torn, les pàgines jsp, treballen sobre tot sobre els servlets controladors del paquet accions i amb les classes definides per representar els objectes de l'aplicació al paquet *classes.*

Seguiment del pla de treball: fites 3, 4 i 5.

Les fites 3, 4, i 5 són un recorregut per tota la implementació del projecte amb les classes java i pàgines JSP necessàries. Seguidament es destaquen a tall d'exemple il·lustratiu algunes solucions optades durant el projecte:

Un cop d'ull al disseny

En el disseny de la part web s'han aplicat per sobre de tot dos patrons de disseny: d'una banda el patró Model / Vista / Controlador per seguir les accions de l'usuari i, de l'altra, el patró Memento aplicat a formularis, per permetre la recuperació de les dades en cas de fallada de la validació de l'entrada. Pel que fa a les classes de programari clàssiques, s'ha utilitzat el patró de disseny de la factoria per implementar les classes responsables de la comunicació amb l'api servidor i la generació d'events i, de l'altra, de les classes que s'encarreguen de la persistència.

En general podem dividir el disseny de l'aplicació entre el cas d'ús més específic de l'entrada al programa (amb la complexitat d'haver de gestionar la part RMI) que ja es tractarà quan parlem de la connexió amb LaCOLLA, la part de repartiment de les peticions fetes en la pantalla de treball principal entre els controladors de cada una d'aquestes; i, finalment, cada cas d'ús de cada acció possible implementat segons el Model /Vista / Controlador.

A part queda tota la gestió d'usuaris, amb la seva pantalla de treball principal i el seu sistema paral·lel de distribució, un altre cop, de les peticions a cada controlador.

Data Access Object: el patró DAO de factories en la persistència

La tria d'implementar la persistència del directori virtual de treball en un fitxer xml és personal i està molt lligada als objectius generals del treball. Ens podem trobar molt fàcilment en el cas de voler migrar cap a un altre model: en una base de dades, per exemple.

Per facilitar aquesta tasca de migració i, en general, per tenir ben separades les responsabilitats entre les classes: fer transparent al model la tasca de la persistència de les dades, se sol utilitzar el patró DAO que, bàsicament, consisteix en la interposició d'una interfície (contracte) entre el model i les dades.

En el nostre cas definim una interfície en el paquet *persistencia* anomenada *PersistenceFactory* sobre la qual treballen les classes encarregades de la lògica de negoci:

```
public interface PersistenceFactory {

    public String getInfo();

    public String getError();

    public boolean crearObjecte(String directoriPare, Objecte obj);

    public boolean eliminarObjecte(String id);

    public boolean copiarObjecte(String directoripare, String id, String tipus);

    public Objecte cercarObjecte(String id);

    public boolean generarAccio(String id, Accio accio, String tipus);

    public Iterator llistarDirectori(String id);
```

```
    public boolean existeix(String id);

}
```

Figura 12 - Serveis de PersistenceFactory

També definim una classe encarregada de generar la factoria segons un paràmetre d'entrada i, finalment, la classe que implementa la interfície i que fa la feina: en el nostre cas *XMLFactory*.

D'aquesta forma, si es vol migrar cap un altre sistema de persistència, només cal modificar la classe generadora de factories perquè inclogui la creació de la nostra segons el paràmetre passat i implementar la interfície segons les nostres necessitats.

Igualment s'ha seguit aquest patró de disseny en el cas de la generació d'events, per si, com es comentarà més endavant, s'acaba creant el servidor de ideaD de forma separada i amb una interfície publicada de treball diferent a l'api de LaCOLLA.

The MVC design pattern

El patró de disseny Model / Vista / Controlador aplicat a programari web és el que s'ha procurat seguir en la implementació de cada cas d'ús. Així, com a exemple, trobem el cas de creació d'una nova carpeta:

Alta d'una nova carpeta

Precondicions

L'usuari s'ha validat al sistema i es troba dins la carpeta pare (carpeta allà on vol generar la nova carpeta) i té drets per crear objectes.

Execució

1) l'usuari omple els camps del formulari d'alta de la nova carpeta i envia la petició;

2) el servlet de control (*servlet/accions/NovaCarpeta*) rep la informació, determina l'acció a realitzar (seguir endavant amb la creació o desfer l'acció), la correctesa de les dades entrades (si se segueix endavant) i, si tot és correcte, instancia l'element encarregat de fer persistent la carpeta;

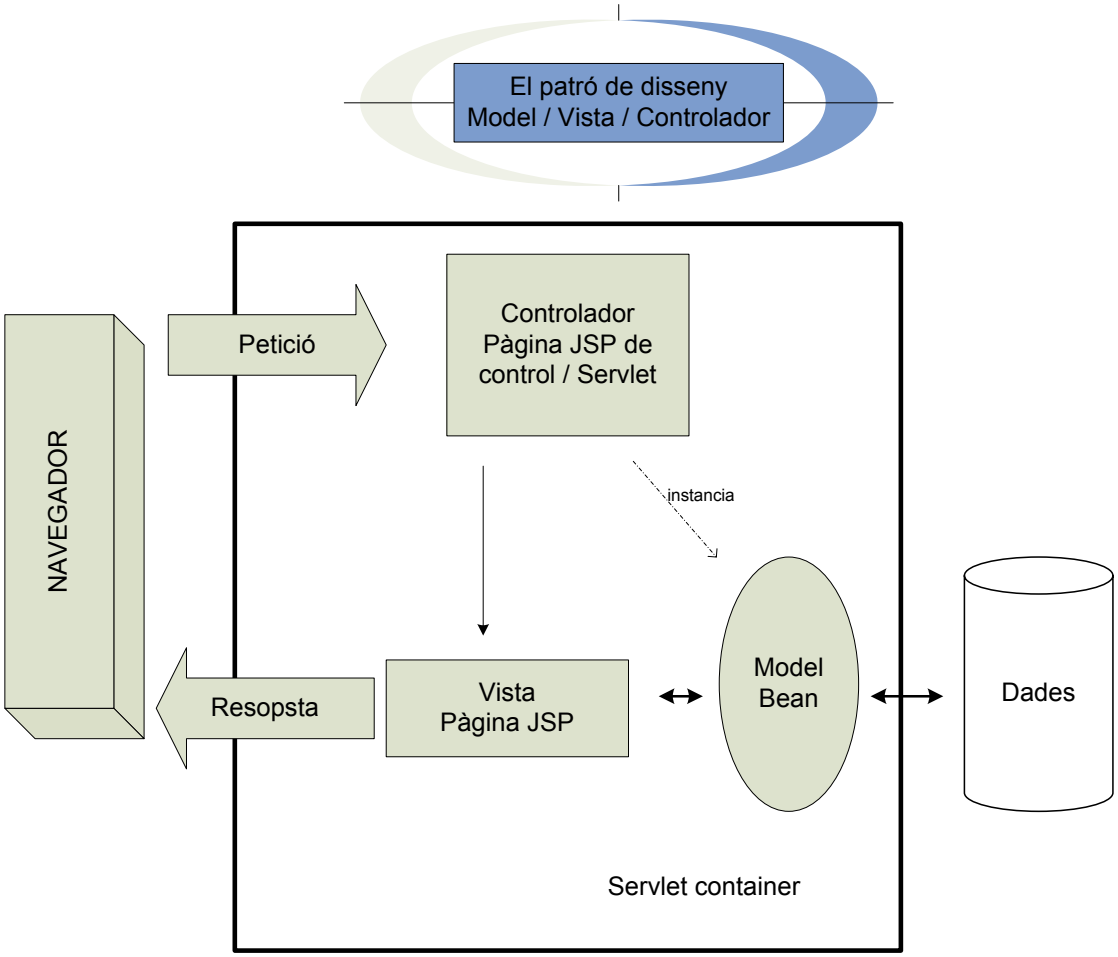

Figura 13 - El patró de disseny Model / Vista / Controlador

3) la persistència s'assoleix instanciant una classe que implementi la interfície *persistenciaPersistenceFactory* que té un servei que rep l'objecte carpeta a crear i l'identificador de la carpeta pare i mira de crear l'objecte. Passa el resultat de la creació al controlador;

4) el servlet de control mostra el resultat de la creació mitjançant la pàgina *formResultatServlet.jsp*.

Post – condicions

La carpeta ha estat creada o bé s'ha informat a l'usuari del perquè no s'ha pogut crear.

En el nostre cas el patró de disseny s'aplicaria:

- Controlador: servlet *NovaCarpeta*;
- Model: classe *Serialitzador*;
- Vista: pàgina *formResultatServlet.jsp*;

The Memento design pattern

Per moltes aplicacions, com també per aplicacions web, resulta molt útil de tenir mecanismes que proveeixin als usuaris de la capacitat de desfer una operació quan viola les regles de negoci especificades. Així, un dels patrons de disseny més repetits al projecte és el Memento. Aquest és un patró de comportament que unit al MVC vist anteriorment permet tractar les dades entrades en un formulari de forma eficient.

L'estructura és la següent:

- Tenim el *'Memento'* que emmagatzema tot o part de les dades originals i l'estat de l'*Originator* i només permet a aquest modificar-les;
- L'*'Originator'* és l'encarregat de crear / anar modificant les dades del *Memento*;
- I el *'Caretaker'* que emmagatzema i cuida dels objectes *Memento* però sense poder-los modificar;

En el nostre cas el formulari d'entrada de dades serà l'originador, ja que és l'únic que crea o modifica les dades del nostre interès; el Memento serà el bean que emmagatzema aquestes dades i el "cuidador" serà la pàgina jsp de control, que crida al bean, no el modifica i només es cuida de que cada cop les dades del formulari s'hi hagin traspassat.

En la següent figura el veiem aplicat al cas d'ús d'alta de nou usuari. En l'aplicació d'aquest patró veiem les grans capacitats de procés de formularis d'una simple pàgina JSP de control amb un bean ben dissenyat.

El contracte dissenyador - programador

La clau del bon funcionament d'aquest patró radica en establir tant les propietats del bean sobre el que es treballarà com els seus noms. Així amb poc o gairebé gens d'esforç es podrà passar la informació entrada en el formulari cap a la lògica de negoci gràcies a la ***introspecció***.

L'exemple: alta d'un nou usuari

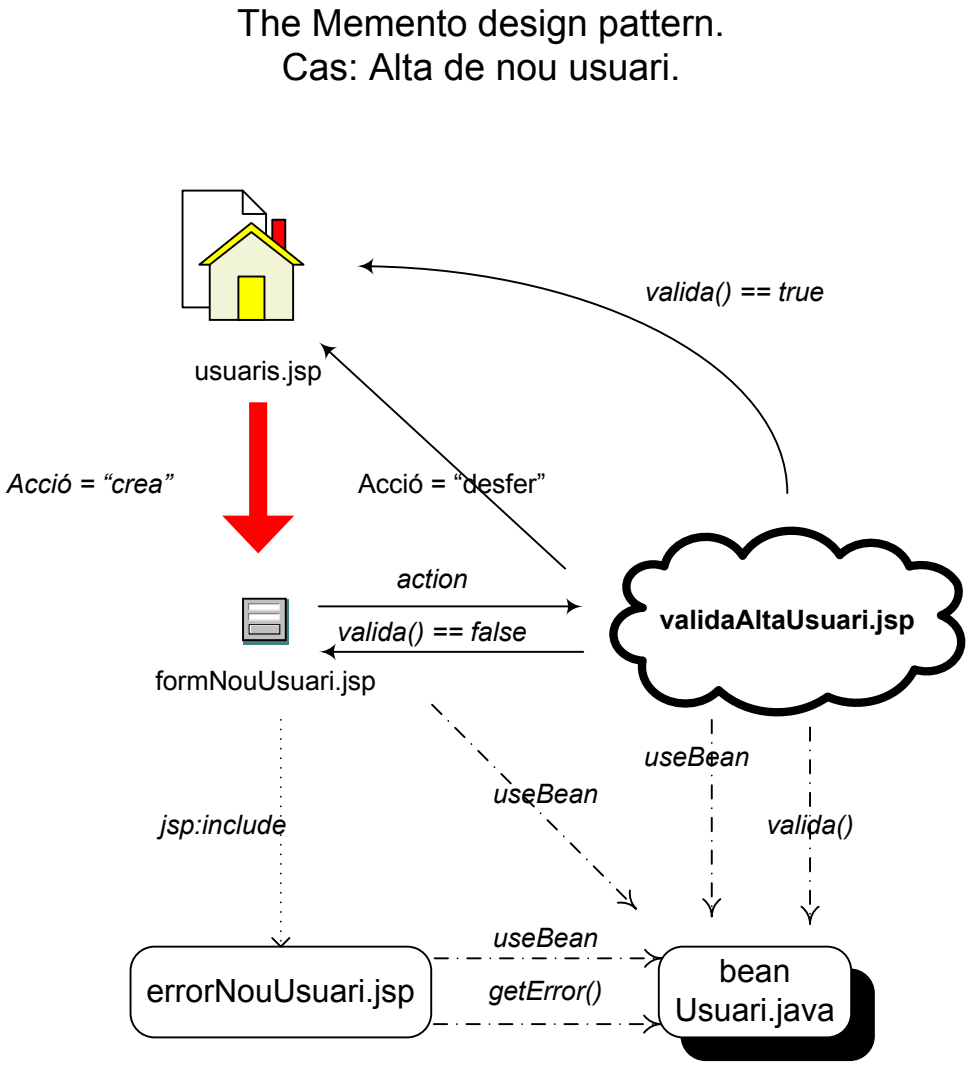

Figura 14 – The memento design pattern

Des de la pàgina principal de treball (per al nostre cas usuaris.jsp) es genera el cas d'ús *"crea"*, que seguint la seva lògica de negoci pròpia acaba responent a la petició ensenyant el formulari d'alta d'usuaris.

formNouUsuari.jsp

Aquesta pàgina comença amb dues accions gens típiques d'un formulari:

1. fa un *include* de la pàgina errorNouUsuari.jsp i

2. instancia el bean Usuari

Més endavant veurem quins avantatges en traiem d'aquest fet, ara només ens interessa saber que la pàgina d'error no mostrarà res i que el bean Usuari, en crear-se (no ha estat mai abans instanciat durant la petició), contindrà totes les seves propietats en blanc. Així dons el resultat que es veurà a pantalla serà un formulari ben corrent.

validaAltaUsuari.jsp

Pàgina JSP de control (en cap cas mostra res a pantalla) que fa el següent:

1. mira si la petició és d'alta o, per contra, s'ha decidit cancel·lar l'acció, cas en el qual retorna a la pàgina de treball inicial;

2. si se segueix amb l'operació d'alta, instancia un altre cop el bean Usuari (ja creat en el formulari, però amb les propietats en blanc);

3. mitjançant introspecció trasllada la informació dels paràmetres de la petició creats pel formulari a les propietats del bean (*set property ='*'*);

4. crida el servei *valida()* que fa les comprovacions necessàries i respon cert/fals segons la correcció de les dades, al seu torn que crea el missatge d'error (si és necessari) com una propietat més del bean;

A partir d'aquest punt arriba la lògica de negoci encapsulada en aquesta pàgina JSP: si el servei valida ha donat per bona l'entrada de dades, la pàgina JSP ens retorna a la pàgina de gestió d'usuaris inicial (evidentment per simplificar no ens estem posant en el tema de la persistència de l'objecte). En cas d'error retorna al formulari d'alta d'usuari.

Cas d'error

Retornem al formulari, però aquest cop la seva aparençà serà diferent. En primer lloc l'incloure la pàgina d'error el que farà serà mostrar-nos el missatge, ja que la pàgina errorNouUsuari.jsp el que fa és instanciar el bean i veure la seva propietat error: en cas de tenir un missatge el mostra a pantalla. En cas contrari la seva sortida és nul·la.

En segon lloc el que veiem és que el formulari porta els camps ja omplerts amb la informació que havíem enviat abans. Ara, en instanciar el bean ja no es crea de zero, sinó que es recupera el processat a validaAltaUsuari.jsp (només hem encadenat pàgines JSP, seguim en la mateixa petició que ha nascut en el formulari inicial) i per tant, amb una mica de programació en java, assolim que els valors per defecte siguin els del bean.

Com veiem el processament de la informació d'un formulari amb una pàgina JSP de control i un bean es fa molt senzilla. El procés d'error pot continuar de forma indefinida fins que el bean dongui el vist-i-plau a les dades entrades per l'usuari.

Persistència: objectes de sessió

La informació que es comparteix entre peticions és la següent:

Identificador	Classe	Explicació
aplicacio	directori.Api.Aplicacio	Objecte que emmagatzema les dades relatives a la connexió, com l'identificador d'aplicació davant LaCOLLA o els serveis per obtenir la referència a l'api de LaCOLLA.
usuari	java.Lang.String	Identificador de l'usuari de ideaD connectat.
login	login.Login	Emmagatzema les dades de l'objecte de connexió de l'usuari.
directori	java.Lang.String	Identificador del directori de treball de l'usuari actual.
copiats	java.util.List	Llista d'elements copiats/retallats.

log	util.Log	Encarregada d'anar creant el log de l'aplicació.
informacio	java.Lang.String	Cadena de caràcters que pot ser usada per passar informació a pantalla a l'usuari.
tipus	java.lang.String	Tipus d'api sobre la qual treballa el client (LaCOLLA o ideaD).

Figura 15 - Objectes de sessió

Fita 6 – Integració amb LaCOLLA

La integració amb LaCOLLA requereix de l'ús de la tecnologia *Remote Method Invocation* (RMI) de Java. Aquesta permet que dues (o més) aplicacions situades en màquines virtuals de Java independents puguin comunicar-se mitjançant un objecte especial anomenat registre.

El cas més habitual és en el qual una aplicació actua de servidora i publica en el registre una *aplication public interface* (api) mitjançant la qual els clients poden sol·licitar-li serveis.

El cas en el què ens trobem és una mica diferent: tenim dues aplicacions que, alhora, col·laboraran, és a dir, alguns cops ideaD demandarà serveis a LaCOLLA i d'altres serà LaCOLLA qui accedeixi a un api publicat per ideaD per notificar.-li informació. Totes dues són alhora servidores i clients, des del punt de vista de l'RMI.

Tasques segons l'arquitectura peer – to – peer per ordinador

Si recordem en l'inici d'aquest document, en la part de l'especificació, s'havien definit dues possibles arquitectures. En aquest primer apartat analitzarem les tasques a realitzar per connectar amb LaCOLLA quan cada ordinador és tractat com a igual. Més endavant, en un apartat específic, ja analitzarem com canviaria la implementació d'aquesta fase si es vol l'estructura parcialment client/servidor.

Així doncs, les tasques amb les qual havíem de fer front en aquesta fase són:

a) Captura de l'api servidora de LaCOLLA;

b) Publicació de l'api servidor de ideaD amb el qual LaCOLLA es comunica amb l'aplicació;

c) Implementació de les respostes d'aquesta api donades les informacions de LaCOLLA;

D'altres tasques relacionades amb aquesta fase incloïen la creació de l'estructura de classes necessària per passar events als iguals.

Però anem a pams:

Estructura de LaCOLLA

LaCOLLA pot funcionar sota diverses implementacions, depenent dels agents que s'instancien en una determinada execució. Per al desenvolupament i proves del projecte s'ha utilitzat una de les configuracions estàndard lliurades amb LaCOLLA amb un *User Agent* (UA), un *Repository Agent* (RA) i un *Group Administration and Presence Agent* (GAPA).

LaCOLLA també disposa dels agents *Task Dispatcher Agent* (TDA) I *Executor Agent* (EA) per proveir serveis d'execució de tasques (temps de CPU) en sistemes distribuïts, però aquests no són necessaris per a ideaD, que només utilitza la part de serveis de repositori d'objectes per als fitxers i disseminació d'events.

Captura de l'api servidora de LaCOLLA

Tot just arrencar, i previ a la finestra de validació d'usuaris (no oblidem que són independents els de ideaD dels de LaCOLLA) l'aplicació instancia la classe directori.Api.Aplicacio que és l'encarregada de processar el fitxer de configuració i, si aquest així ho indica, establir la connexió amb LaCOLLA. Aquest objecte és passat a la sessió de l'usuari amb l'id d'aplicació respost per LaCOLLA.

Publicació de l'api de ideaD

La classe *ApplicationsSideApiImpl* no compila

El primer petit entrebanc va venir derivat que la classe que implementava l'api (amb els serveis en blanc, només de prova) no compilava. El problema, com al final es va resoldre, provenia del servei ***newEvent(String, Event)*** que no deixava crear correctament la classe que implementava aquesta interfície: en compilar sempre deia que o bé s'havia de definir la classe com a abstracta o bé definir aquest servei (quan ja estava). Finalment, es va haver de tirar endavant definint el servei per un objecte *LaColla.core.data.Event* i un altre per *LaColla.core.data.app.Event*.

Impossibilitat de publicar l'api del client

El primer dels greus problemes als quals es va haver de fer front i que ha retardat considerablement el projecte és un error de tipus UnmarshallException amb el missatge *'no protocol'* que es llençava en el moment de publicar l'api client. Evidentment l'error no té sentit, ja que el protocol en rmi és per defecte *rmi.* El més curiós del cas és que el mateix codi executat des d'una classe java funcionava correctament, mentre que des d'un servlet no funcionava.

En un primer moment es va creure que podia estar relacionat amb un bug detectat en el servidor Apache – Tomcat quan aquest s'instal·la en sistemes operatius Windows en un directori amb espais en blanc en el seu nom. Aquesta errada va ser suggerida pel mateix consultor com a possible explicació del problema, així que es va procedir a desinstal·lar el servidor web i, aprofitant l'ocasió, instal·lar l'última versió del Tomcat en un directori sense espais en blanc.

El fet és que després d'això el codi seguia sense funcionar. Finalment es va creure que podia estar relacionat amb el plug-in de Sysdeo per Eclipse que s'utilitzava per desenvolupar el codi: aquest permet treballar sense desplegar el codi directament al servidor: crea un xml a Tomcat_home/conf/Catalina/localhost amb el nom del context path del projecte i el directori de treball dins el workspace de l'Eclipse. Això es va creure que podia despistar el *Class Loader* de l'RMI d'on cercar les classes *stub* (d'aquí el tipus d'excepció: *Unmarshall*). Efectivament, en desplegar-se el projecte dins del directori webapps va desaparèixer l'error.

El silenci de LaCOLLA

Aquest ha estat el punt mort on ha caigut el projecte. Malgrat que ja no donés error la publicació de l'api de client, se seguien sense rebre events de LaCOLLA. En analitzar la sortida d'aquesta es va detectar el següent error:

```
java.rmi.NotBoundException: /AplicationsSideApi
 at sun.rmi.registry.RegistryImpl.lookup(RegistryImpl.java:106)
 at sun.rmi.registry.RegistryImpl_Skel.dispatch(Unknown Source)
 at sun.rmi.server.UnicastServerRef.oldDispatch(UnicastServerRef.java:375)
 at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:240)
 at sun.rmi.transport.Transport$1.run(Transport.java:153)
 at java.security.AccessController.doPrivileged(Native Method)
 at sun.rmi.transport.Transport.serviceCall(Transport.java:149)
```

```
at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:460)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:701)
at java.lang.Thread.run(Thread.java:595)
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(Unknown Source)
at sun.rmi.transport.StreamRemoteCall.executeCall(Unknown Source)
at sun.rmi.server.UnicastRef.invoke(Unknown Source)
at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
at LaColla.Api.LookUp.<init>(LookUp.java:20)
at LaColla.core.data.Application.newConnectedMember(Application.java:48)
at LaColla.core.data.ApplicationsInfo.newConnectedMember(ApplicationsInfo.java:76)
at LaColla.core.components.UA.doAcceptAuthenticationOfParticipant(UA.java:793)
at LaColla.core.util.msgServiceUA.run(msgServiceUA.java:140)
```

Pel tipus d'error primer es va creure que l'api no era publicat, però si es llançava des de l'Eclipse en paral·lel una aplicació que connectés amb el registre en el port especificat i fes un *Naming.list()* el resultat era l'esperat: *AplicationsSideApi*.

Més tard es va pensar que no fos un problema del *ClassLoader* que no tingués accés als arxius *class* del projecte. Així doncs es va procedir a generar un arxiu *jar* que inclogués la classe *stub*, afegir-lo als jars de LaCOLLA i modificar el *batch* d'arrencada d'aquesta per incloure'l en el *classpath*. Res no es va resoldre.

Encara que en teoria amb això no hi hauria d'haver cap problema amb la càrrega de classes, per si de cas, es va procedir a afegir el codi necessari abans de publicar l'api per arrencar una instància de *RMISecurityManager* per si eren problemes de seguretat. En principi no hauria de ser aquest el problema ja que el gestor de seguretat només és necessari si els clients s'han de descarregar codi des d'ubicacions diferents al seu *classpath*. Evidentment, el codi seguia sense funcionar.

Per seguir amb el tema (no deixar possibilitats) es va decidir arrencar manualment el registre de RMI havent afegit la propietat *codebase* apuntant directament a l'arxiu *stub* del projecte desplegat al Tomcat i creant un arxiu de

política de seguretat que ho permetia tot, tant al directori base de Java com al de l'usuari i l'error seguia igual. Fins i tot es va arribar a crear una classe gestora de seguretat pròpia sense èxit.

Cal afegir que es treballava amb el *firewall* del *service pack 2* de Windows desactivat i que s'han provat altres aplicacions que funcionen sobre el registre de RMI (clàssiques client/servidor) i han funcionat en els mateixos ports.

Implementació d'ApplicationsSideApi

Evidentment en aquest punt, donades les circumstàncies, hi ha poc a explicar, llevat de les idees.

El primer pas consistia en generar un conjunt de classes d'event consistents per reproduir les accions necessàries:

Events

Tal com s'ha comentat, s'ha generat un paquet que estén els events disponibles a LaCOLLA, anomenat *xLaColla.event* on hi podem trobar les classes que possibiliten la comunicació dels esdeveniments d'ideaD.

La classe *EvtObjectIdeaD*

Recordem que les classes que representen els objectes que tracta ideaD (carpetes, fitxers i notes) estan dins el paquet *Classes* i que totes implementen la interfície *java.io.Serializable*, és a dir, les podem enviar com a objectes remots.

Però a més a més, totes implementen la interfície definida en el propi programari *Objecte*, no només aprofitant que part dels serveis que ofereixen són comuns entre els objectes (tots tindran una data de creació i un identificador únic, per exemple) sinó que també per certes accions serà independent l'objecte que es tracti i podrem aprofitar el polimorfisme.

Per això en aquesta classe s'hi ha afegit els atributs usuari (usuari de ideaD, no de LaCOLLA), idPare (identificador de l'element pare del qual, en l'arbre XML, hauria de penjar un nou element) i element que és del tipus *Objecte*.

A més s'ha decidit utilitzar els atributs ja definits de la següent forma: *eventId* indicarà el tipus d'event, és a dir, si "CREA" o "MODIFICA" o bé "ELIMINA" mentre que *event* informarà del tipus d'objecte que s'ha traspassat: "arxiu", "carpeta" o "nota".

La classe *EvtAccioIdeaD*

Molt similar a l'anterior, però amb un objecte de tipus *Accio* enlloc de l'element *Objecte*. S'usa per passar les accions de llegit o vist dels usuaris sobre els objectes.

La classe *EvtUsuariConnectat*

Com que s'ha separat els usuaris de ideaD dels de LaCOLLA, tenim uns events diferents en el loginar-se. Només s'afegeix l'atribut usuari de ideaD i el camp ja existent de *eventId* per marcar si es connecta o desconnecta.

Reacció davant dels events (implementació)

El programari havia d'actuar de quatre formes diferents davant d'un nou event:

- Omitint-lo (segurament hi ha events que llença LaCOLLA que no són del nostre interès);
- Mostrant informació a l'usuari (cas d'un nou membre de ideaD connectat);
- Realitzant accions (crear una nova carpeta en l'estructura de treball);
- Realitzant accions i mostrant alhora informació a l'usuari (un usuari diu que ha vist o llegit un dels fitxers i s'informa als membres del grup connectats, a més de modificar el fitxer que manté l'estructura perquè això consti en la informació de l'arxiu);

Quan el tractament previst per un event és mostrar alguna informació a l'usuari, aquesta es fa present a pantalla mitjançant un atribut de sessió. Així, cada cop que l'usuari actualitzi la pantalla de treball (es fa quan realitza una acció) veurà acumulada tota la informació que se li ha enviat i en l'acció actualitzar es buida l'element.

En el cas de modificacions de l'estructura de treball, es cridaran les classes responsables de la serialització i se'ls demanarà l'acció.

L'objectiu final de l'estructura client/servidor per una organització

Donat que no s'ha pogut completar ni el primer prototipus pel problema comentat, aquesta fase queda completament en disseny. Tot i així, podem comentar les idees sobre les que es volia treballar:

D'on va sorgir la idea

Des de bon principi l'autor va plantejar-se el projecte veient LaCOLLA com un proveïdor de serveis sobre (i gràcies a) la qual requeien els requeriments de la distribució i descentralització, és a dir, la clau de volta de l'arquitectura peer – to – peer; mentre que ideaD només era una aplicació web que se'n beneficiava.

A més, resultava difícil d'acceptar el fet que cada igual hagués de tenir tota la instal·lació: LaCOLLA, més el servidor web per ideaD, més ideaD per poder treballar. S'estava obligant a treballar amb una tecnologia (web) amb les seves restriccions i problemàtiques sense aprofitar-ne el principal dels beneficis: la utilització per clients amb només un navegador instal·lat. Si de debò s'havia de tenir instal·lat sencer ideaD, no era millor desenvolupar-la com una aplicació clàssica?

Arran de tots aquests pensaments va néixer la necessitat de donar una nova visió a les aplicacions web que es connectessin a LaCOLLA.

Els problemes de l'arquitectura peer – to – peer per organització

LaCOLLA treballa publicant una interfície pública en un port determinat (als fitxers de configuració d'aquesta) on espera rebre les peticions dels programes clients via Remote Method Invocation. Al seu torn, per retransmetre els events espera que el programari client publiqui una altra api on escoltarà els seus missatges.

En una arquitectura peer – to – peer clàssica, com que cada igual tindrà una instància de LaCOLLA es publicarà la seva api de recepció d'events; però en aquesta arquitectura parcialment client/servidor només podrà publicar-se una api per servidor web. Així doncs, la classe *ApplicationsSideApiImpl* haurà de responsabilitzar-se de capturar tots els events i notificar només als clients – web interessats.

Publicació de l'*ApplicationsSideApi*

Com que només s'ha de tenir una api d'aplicació publicada per tots els clients, en un primer moment es va pensar resoldre el tema deixant la responsabilitat de la publicació de la interfície al servei init() d'un servlet i la seva despublicació (unbind) al destroy(), però de seguida ens vam adonar que no controlàvem quan el contenidor de servlets pogués decidir destruir la instància del servlet.

Una altra idea era aprofitar l'objecte *Web Context* compartit per tots els servlets, per a dipositar-hi les referències comunes, però seguia suscitant certes dificultats i impedia el creixement de l'aplicació, ja que implicava un únic servidor web per tots els clients.

La solució s'allunyava del món web i era més fàcil del que semblava: crear un servidor per ideaD.

El servidor de ideaD

Aquest, en el moment d'arrencar-se, faria totes les tasques necessàries per gestionar la connexió amb LaCOLLA: carregar el paràmetres de la configuració, obtenir la referència a l'api de LaCOLLA, publicar un únic api d'ideaD per tota la organització i loginar-se a LaCOLLA amb l'usuari corporatiu.

Per als clients publicaria també per RMI el seu api propi, amb el qual aquests interaccionarien. Aquesta interfície hauria de permetre les operacions de loginar-se a ideaD, sortir de ideaD i totes les accions necessàries per treballar amb LaCOLLA, però ocultant-ne la complexitat.

Allò que volem dir és que el client no té cap necessitat de conèixer l'id d'aplicació o de grup que es manté amb LaCOLLA. Això és responsabilitat del servidor de ideaD, però molts cops ens és demanat (per crear events, per exemple). Així els clients tindrien una api pròpia on segons la petició que fessin el servidor s'encarregaria de traduir-la per passar-la a LaCOLLA. Així els clients només tindrien visible la interfície que necessiten.

Pas d'informació des de l'ApplicationsSideApiImpl

Els clients web farien peticions mitjançant l'api del servidor de ideaD, aquest les traduiria i traspassaria (si fos necessari) a l'api de LaCOLLA i aquesta respondria, utilitzant l'api de client de LaCOLLA publicat pel servidor de ideaD.

Queda el pas de fer saber als clients de ideaD els esdeveniments llençats per LaCOLLA. Està clar que podem seguir amb el model i fer que publiquin la seva api pròpia, però aquesta solució es creu massa complexa, ja que, bàsicament, la informació que es necessita des del client és les actualitzacions de l'estructura virtual de fitxers que pot anar fent el servidor per si sol i el pas d'alguns missatges, és a dir, objectes String que els clients podrien obtenir fent crides periòdiques a un servei de l'api del servidor de ideaD.

El que caldria és que el servidor mantingués aquest objecte per cada grup o usuari de ideaD.

Estructura d'apis a l'RMI amb ideaD

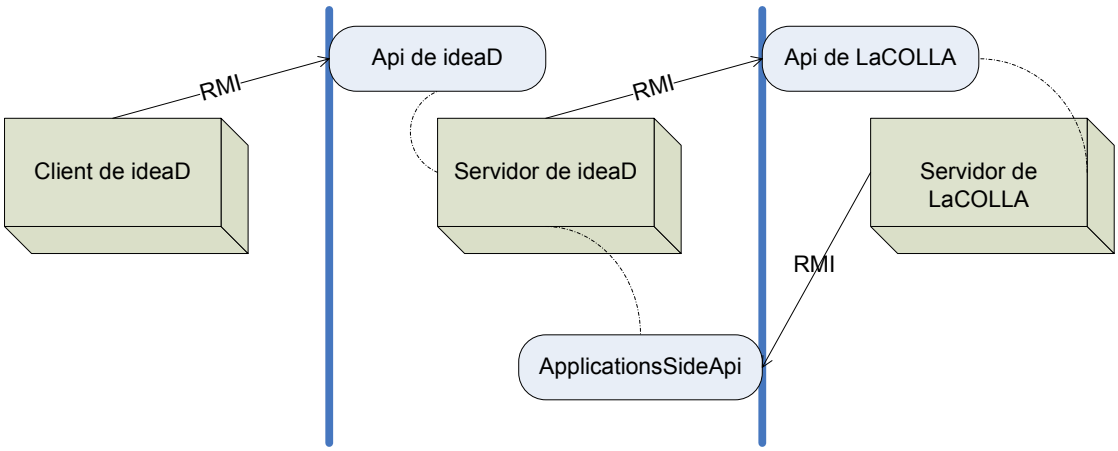

Figura 16 - El servidor de ideaD

El pas d'arxius

Un altre punt difícil, ja que la solució més senzilla és fer-lo en dues fases: passant pel servidor de ideaD, fet que duplicaria el cost de l'operació.

Canvis en el projecte

Els canvis en el projecte per migrar a l'altra estructura implicarien una entrada diferent (s'havia pensat en tenir dues pàgines JSP diferents) que canviaria el procés del loginat i la sortida (actualment s'invalida la sessió i es fa logout de LaCOLLA, quan aquest darrer punt no caldria).

Pels events, com que ja s'ha pensat en el tema fent que les classes que els implementen segueixen el patró del *DAO Factory*, només caldria crear una classe que tractés l'api de ideaD enlloc de l'api de LaCOLLA carregat al principi.

Producte

Es lliuren amb aquest document un arxiu web (war) amb el prototipus de ideaD per funcionar amb l'arquitectura peer – to – peer per ordinador per a desplegar en el servidor, així com un arxiu java (jar) que s'ha de copiar en algun directori i fer-lo accessible en el *classpath* de LaCOLLA i un arxiu comprimit (zip) amb dades de configuració.

Instal·lació i configuració

Abans:

S'ha de tenir una instal·lació de LaCOLLA, amb una màquina virtual de Java (es recomana Java 5 perquè és sobre el qual s'ha desenvolupat el projecte) i un servidor web amb extensió per interpretar servlets i pàgines JSP.

Instal·lació:

Agafar l'arxiu war proveït i desplegar-lo al servidor, ja sigui de forma manual o bé, si és el cas, deixant-lo a la carpeta de descompressió automàtica que proveeixi el servidor.

Crear una carpeta a l'arrel del directori c: anomenada pTFC i descomprimir-hi l'arxiu zip, on hi ha d'haver, com a mínim, un fitxer config, un fitxer anomenat dir.xml i un fitxer d'usuaris i un de grups.

Copiar l'arxiu jar en alguna ubicació (si es vol a la mateixa carpeta) i modificar les dades d'arrencada de LaCOLLA perquè aquest figuri en el seu *classpath*.

Configuració:

Editar el fitxer config per triar si es vol començar ja connectant a LaCOLLA (local=N), l'usuari i contrasenya (pwd) a utilitzar per connectar amb LaCOLLA, la màquina (host) i el port on cercar l'api de LaCOLLA, així com el nom amb què està publicat (name), el mateix per l'api de client de LaCOLLA (host_cli, port_cli i name_cli), l'identificador del grup de treball a LaCOLLA, el mateix per l'agent GAPA amb la màquina i el port on escolta, la ubicació dels fitxers d'usuaris i grups (descomprimits abans a c:\pTFC i que, per tant, no caldria canviar) i el tipus de persistència (xml per defecte).

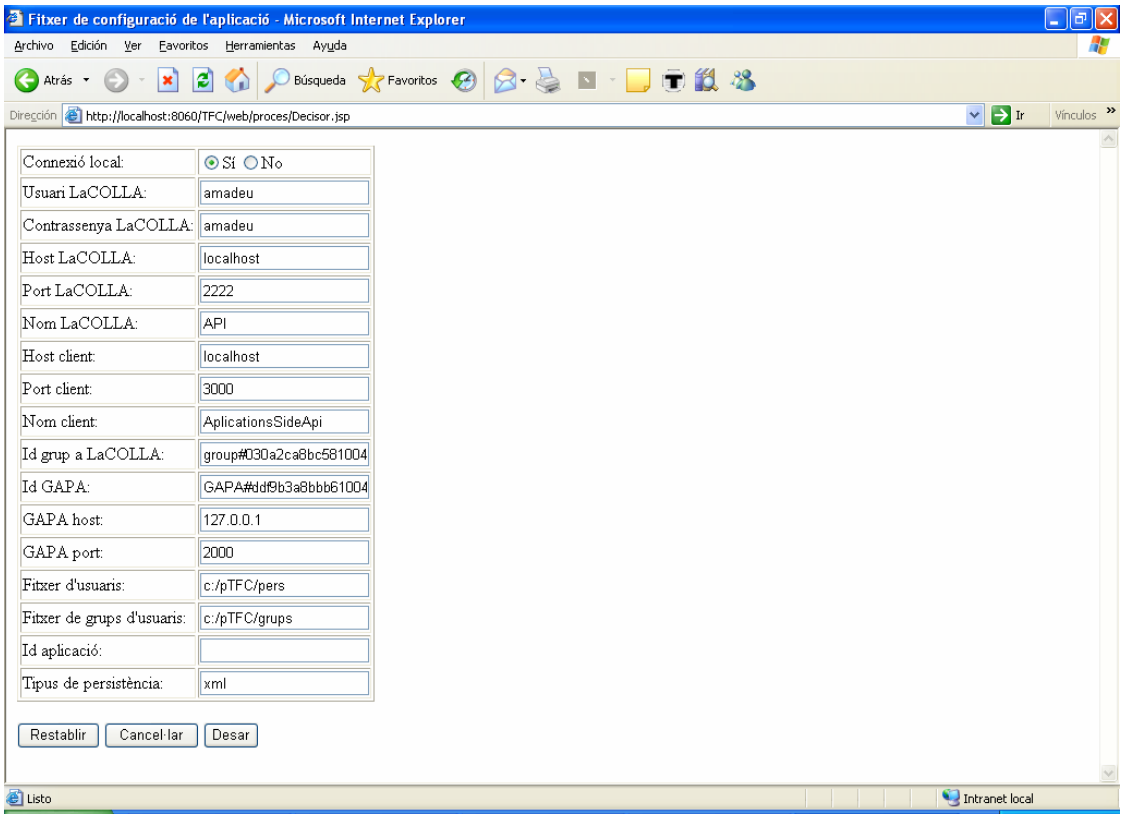

Figura 17 - Edició del fitxer de configuració des de ideaD

Connexió:

Recordem que els usuaris de ideaD són propis. Per defecte es deixen donats d'alta l'usuari administrador (id = Sistema i pwd = root) i un usuari sense privilegis d'administració d'usuaris (id = usuari i pwd = usuari). Sense unes dades correctes no es pot entrar a l'aplicació.

Funcionalitats implementades

Si recuperem la idea inicial del projecte mostrada en el pla de treball inicial, un cop desenvolupades les classes base del projecte i la seva serialització en el fitxer xml, es procediria a implementar les funcionalitats del projecte primer, de forma independent amb LaCOLLA i, un cop es verifiqués el funcionament esperat, diguem "en local", es passaria a tractar el problema de LaCOLLA.

Per facilitar aquest tipus de treball el mateix fitxer de configuració del programari té una opció que permet evitar qualsevol acció de connexió amb LaCOLLA,

donant així coma resultat un programari que funciona de forma totalment independent de LaCOLLA i un altre que s'hi connecta. Evidentment en el primer cas queda coix, ja que no es disposa de la persistència dels objectes fitxers ni, tampoc, de les facilitats de LaCOLLA per traspassar events.

Prototipus independent de LaCOLLA

Llistat de funcionalitats implementades:

- ✓ Login: validació de l'usuari en el repositori d'usuaris de ideaD i configuració de la finestra de treball segons el seu perfil (administradors, amb gestió d'usuaris completa, usuaris amb només canvi de la pròpia contrasenya).
- ✓ Logout: sortida segura amb confirmació i invalidació de la sessió de treball.
- ✓ Navegació entre carpetes
- ✓ Barra de navegació directa: barra de navegació on es mostra el camí fins l'arrel i es pot utilitzar per saltar a qualsevol dels nivells anteriors;
- ✓ Salt a l'arrel: salt directe a l'arrel des de qualsevol directori;
- ✓ Nova carpeta: creació d'una nova carpeta;
- ✓ Nova nota: creació de la nova nota;
- ✓ Respondre a nota: creació d'una nova nota com a resposta d'una d'anterior;
- ✓ Donar per llegit: marca que l'usuari actual ha llegit el fitxer en qüestió. Aquesta informació es mostra sempre en demanar la informació completa relacionada amb l'objecte;
- ✓ Donar per vist: marca que l'usuari actual ha vist el fitxer en qüestió. Aquesta informació es mostra sempre en demanar la informació completa relacionada amb l'objecte;
- ✓ Eliminar: esborra (sense paperera de reciclatge) els objectes triats. Es demana confirmació però de moment no s'ha implementat un sistema segur d'esborrat sense que no estigui en ús per un altre usuari. No es permet eliminar carpetes (no s'ha implementat la forma de validar el contingut) ni fitxers (per no descoordinar la informació amb LaCOLLA);

- ✓ Copia: afegeix els objectes triats a la llista d'elements copiats amb la marca de només copiar, per ser posteriorment deixats en un altre directori;
- ✓ Retalla: fa el mateix que copia però amb la marca de retallar perquè en l'acció d'enganxar elimini;
- ✓ Enganxa: deixa els fitxers en el directori actual i, en el cas de retalla, els elimina de l'original;
- ✓ Cancel·la copia / retalla: desfà les accions de copia i retalla que no s'hagin enganxat
- ✓ Veure informació de l'objecte: difereix en el cas de l'objecte, però inclou qui ha vist/llegit el mateix o el contingut de la nota;
- ✓ Canvi de contrasenya: modifica el password de l'usuari;
- ✓ Gestió del fitxer de configuració: permet fer canvis en la configuració de ideaD;
- ✓ Nou usuari: creació d'usuaris;
- ✓ Veure informació de l'usuari: veure les dades relatives al grup, rol o si està actiu (es permet validar-se al programari a l'usuari de l'id que es consulta);
- ✓ Modificar usuari: canviar dades relatives al grup al qual pertany, el seu rol o si està actiu;
- ✓ Eliminar usuari: esborrar l'usuari;
- ✓ Crear grup: crear nou grup de treball d'usuaris
- ✓ Eliminar grup;
- ✓ Fitxer log: bona part de les accions queden marcades en un fitxer log de quan es van fer i els motius de la tirada enrere o bé errors;

Prototipus connectat a LaCOLLA

Té totes les accions anteriors implementades més la càrrega i descàrrega de fitxers amb el greu inconvenient de no rebre notificacions dels events de LaCOLLA.

Finestra principal de treball

La presentació del programari és un altre aspecte que ha quedat sense temps de desenvolupament, donat que es considerava cabdal mirar de rebre els events de LaCOLLA i s'ha estat dedicat a aquest tema fins a darrera hora.

La pàgina principal de treball és com la veiem en la figura. Dividida en tres parts, a l'esquerra els botons d'accions, al centre i el què ocupa més espai el llistat dels directoris i arxius de la carpeta de treball. A sota, l'espai d'informació i fitxers en tràmit de ser copiats.

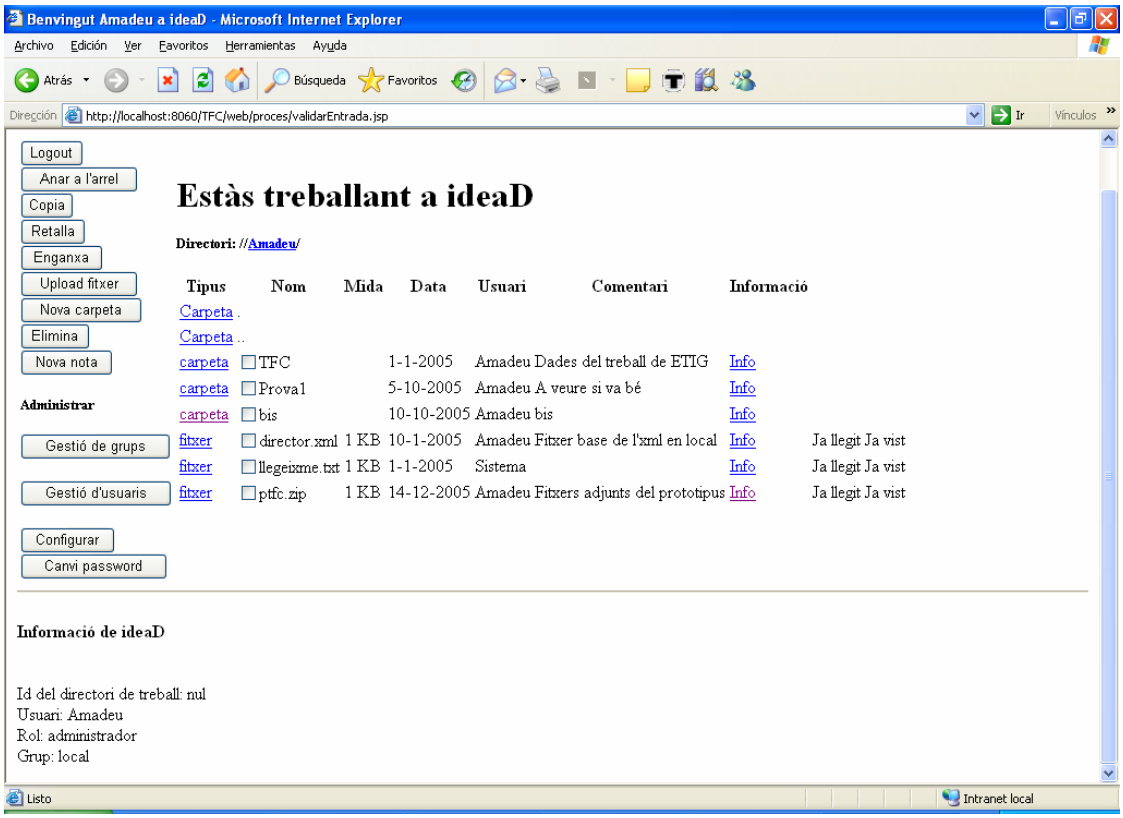

Figura 18 - Finestra principal de treball

S'han dividit les accions que es poden realitzar sobre conjunts d'objectes, que s'han implementat en forma d'accions de formulari i caixes de selecció, de les que només es poden realitzar individualment sobre un objecte, que s'executen per anclatges d'html cap als seus controladors (veure informació, navegar cap a una carpeta o descarregar un fitxer)..

Les accions que requereixen de noves dades a entrar per l'usuari acaben en formularis clàssics com el que es veu.

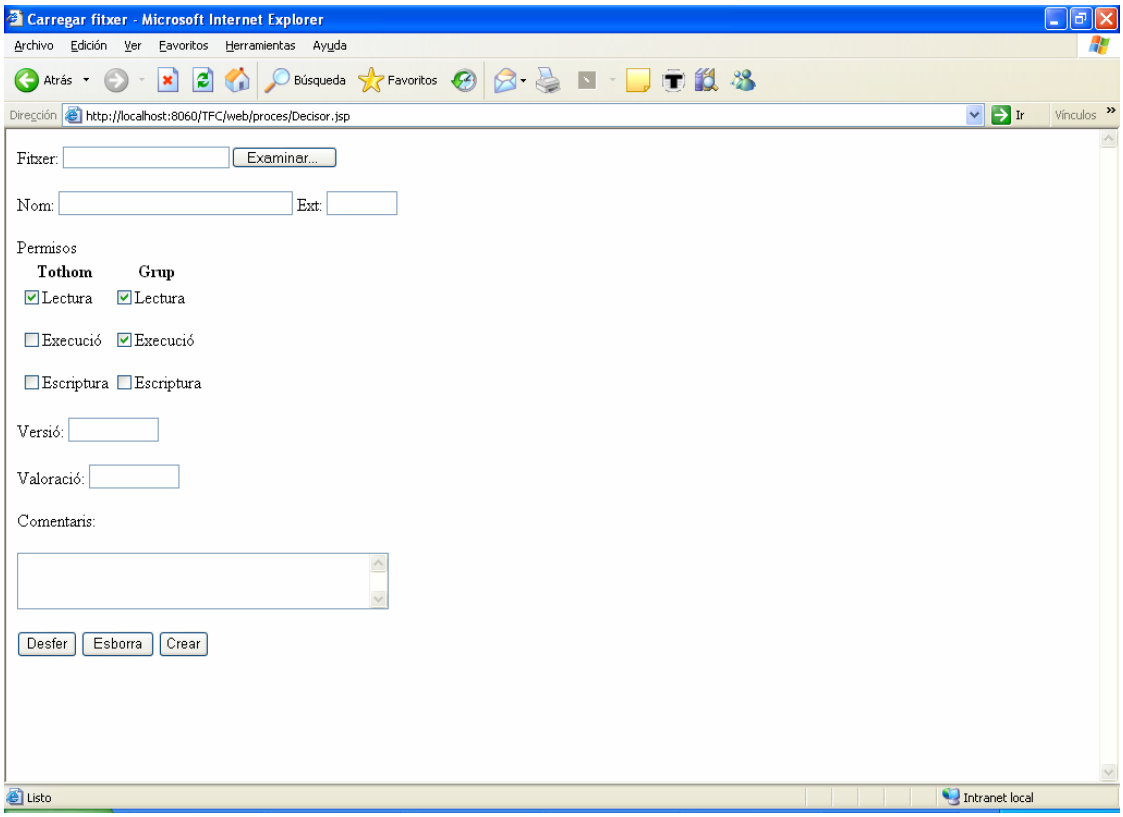

Figura 19 - Formulari de càrrega de fitxer

Rols d'usuaris

Segons l'usuari que entra al programari, la part de tasques de la gestió d'usuaris permet utilitzar la gestió d'usuaris integral o només canviar de contrasenya; perquè s'han definit dos rols diferents: administrador (inclou gestió d'usuaris) i usuari.

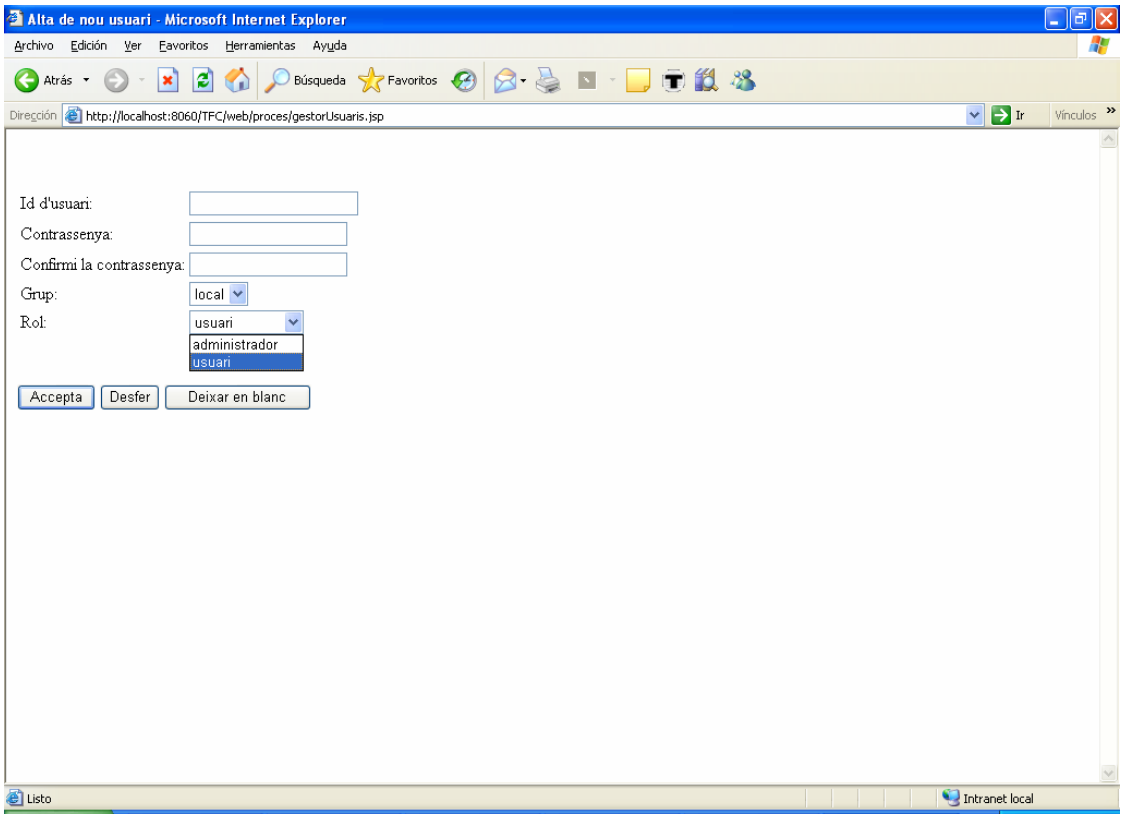

Figura 20 - Administració d'usuaris

Millores proposades

Entorn i presentació

Evidentment l'entorn actual és força "lleig" i caldria millorar-lo, però com es pot veure en el pla inicial, no es va preveure temps per tractar-lo, de forma conscient, ja que allò que, per damunt de tot, es pretenia era crear un producte sòlid i si es trobava temps, anar dedicant-lo a la presentació però donats els grans inconvenients trobats en el desenvolupament, aquest temps no ha existit.

D'altra banda també serien interessants altres millores – accions de la presentació, com ara permetre a l'usuari ordenar els fitxers per diversos criteris (mida, data, etc...) o bé en el cas de les notes presentar-les en estructura arbòria com qualsevol fòrum, seguint camins de les respostes.

Gestió d'usuaris

Per implementar una gestió d'usuaris entenem dos fets:

En primer lloc aplicar els criteris de validació d'accions d'usuaris sobre objectes ja comentats. A tall d'exemple impedir que un usuari sense drets de lectura sobre un fitxer se'l pogués descarregar.

En segon lloc posar en funcionament el sistema de bloquejos per garantir la coherència d'accions com eliminar.

Versionat de fitxers (i reemplaçar arxius en general)

Com s'ha vist l'estructura de fitxers permet emmagatzemar la versió d'aquestos (codi a entrar per l'usuari). Caldria implementar un sistema que permetés reemplaçar un fitxer per un altre però mantenint-los els dos descarregables fins a nova ordre (eliminació d'alguna versió).

Actualment es pot fer d'una forma poc elegant, ja que el programari no valida si es creen diversos arxius amb el mateix nom: li és igual perquè internament els reconeix amb l'identificador únic generat per LaCOLLA o bé, si són carpetes o notes, pel mateix ideaD.

Eliminar objectes mitjançant la paperera de reciclatge

Tot acabant el projecte ens vam adonar que, malgrat representava més feina i per això s'havia descartat incloure-ho en el treball inicial, el sistema de

funcionament de LaCOLLA i el seu sistema distribuït anaven a la mida de l'eliminació d'objectes mitjançant una paperera de reciclatge.

El quid de la qüestió romandria en el fet que, després de realitzar una eliminació, l'objecte passaria a la paperera de reciclatge. L'eliminació seria provisional i no afectaria a les dades emmagatzemades a LaCOLLA ni es podria eliminar definitivament fins, per exemple, passat un termini de temps que es podria fixar per garantir que ningú no l'usa o causa incoherències.

D'una manera o altra esperaríem a que l'event de l'eliminació provisional s'hagués propagat pels iguals sense rebre'n cap resposta indicant el contrari (desautoritzant l'eliminació) abans de permetre que pugui ser donat de baixa permanentment.

Cercador d'objectes

Una eina molt útil en tot sistema de fitxers que et permeti fer cerques per diversos criteris en un directori i tota la seva estructura de fitxers, notes o carpetes.

Missatgeria

Seria interessant incorporar un sistema similar al del BSCW per informar als iguals via correu electrònic d'aquells fets que això ho haguessin configurat.

Bibliografia

- Marquès, J.M. [2003] : LaCOLLA: una infraestructura autònoma i autoorganitzada per facilitar la col·laboració. http://lacolla.uoc.edu/lacolla;
- Sun Microsystems, Java Blue Prints, http://java.sun.com/reference/blueprints/;
- Apache Tomcat Documentation, http://tomcat.apache.org;
- Basic Support for Cooperative Work (BSCW) http://bscw.fit.fraunhofer.de;
- IBM Lotus Domino, http://www-142.ibm.com/software/sw-lotus/products/product4.nsf/wdocs/dominohomepage;
- The Java2 Enterprise Edition 1.4 Tutorial for Sun Java System Application Server;
- Tutorial, jGuru, Remote Method Invocation, http://java.sun.com/developer/onlineTraining/rmi/index.html;
- Tutorial, jGuru, JavaServer Pages Fundamentals, http://java.sun.com/developer/onlineTraining/JSPIntro/contents.html;
- Java Servlet API Specification;
- JavaServer Pages Syntax Reference;
- Govind Seshadri, Advanced form processing using JSP, http://www.javaworld.com/javaworld/jw-03-2000/jw-0331-ssj-forms.html;
- Internet Assigned Numbers Authority, MIME Media Types, http://www.iana.org/assignments/media-types;
- Tecnologia servlet: http://java.sun.com/products/servlet/whitepaper.html;
- Model vista controlador (MVC): http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/web-tier/web-tier5.html;