Universitat Autònoma
de Barcelona

MONITOR

DE SERVIDORS JMS

Memòria del projecte

d'Enginyeria Tècnica en

Informàtica de Gestió

realitzat per

Jordi Manzano Ulloa

i dirigit per

Marc Talló Sendra

Escola Universitària d'Informàtica

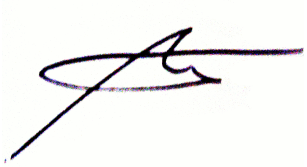
Sabadell, setembre de 2010

El/la sotasignant, Marc Talló Sendra,
professor/a de l'Escola Universitària d'Informàtica de la UAB,

CERTIFICA:

Que el treball al que correspon la present memòria
ha estat realitzat sota la seva direcció
per en Jordi Manzano Ulloa

I per a que consti firma la present.
Sabadell, Setembre de 2010



Signat: Marc Talló Sendra

La tecnologia Java Message Service és una API que habilita la transmissió de missatges entre aplicacions Java, actuant com a *middleware* i fent possible el funcionament d'aplicacions distribuïdes sobre plataformes heterogènies.

El monitor de servidors JMS és un projecte basat en el disseny i implementació d'una eina GUI, destinada a programadors i equips de proves, que treballin amb la tecnologia J*ava Message Service,* multiplataforma i multiservidor, que podrà monitoritzar un nombre variat de servidors JMS des de qualsevol sistema que tingui una màquina virtual de Java instal·lada.

L'aplicació té com a principal objectiu visualitzar de forma clara i senzilla l'estat global d'un servidor JMS, mostrant les cues i tòpics creats, juntament amb la possibilitat de realitzar accions sobre les mateixes destinacions (enviament i eliminació de missatges residents al servidor) i la creació de gràfiques sobre el tràfic de missatges. Així doncs, el monitor ha d'esdevenir una eina focalitzada al tràfic de missatges, que permet a simple cop d'ull veure informació del servidor sense interactuar gairebé amb la mateixa.

Index

CAPÍTOL 1 Introducció

1.1 PRESENTACIÓ

La Java Message Service (JMS) és una API[1] (interfície de programació d'aplicacions) de l'empresa *Sun Microsystems*, que habilita la transmissió de missatges entre aplicacions desenvolupades sota la plataforma Java 2. Actua doncs com *middleware orientat a missatges* (MOM); basat en una infraestructura d'enviament i recepció de missatges, augmentant la portabilitat i flexibilitat del sistema i fent possible el funcionament d'aplicacions distribuïdes sobre plataformes heterogènies. Per la comunicació s'utilitzen cues o tòpics:

- CUES: La comunicació s'estableix punt a punt; un client productor de missatges envia missatges a una cua. Un client consumidor es connecta a aquesta per tal d'obtenir missatges que en ser consumits, seran eliminats de la cua. Aquest model assegura la persistència del missatge en cas de no ser consumit.

1 **API** (Application Programming Interface) és el conjunt de procediments i funcions que ofereix una llibreria per ser utilitzada per un altre programa com una capa de abstracció.

- TOPICS: Es basa en un model publicador / subscriptor: existeixen diversos clients que publiquen missatges mitjançant el tòpic, que podran ser consumits al mateix temps per diversos clients subscriptors. Aquest model no assegura la persistència del missatge en cas de que no hagi subscriptors escoltant, provocant doncs la pèrdua de les dades que contenia el missatge.

Cal dir que la implementació de la interfície JMS vindrà donada per un proveïdor de missatgeria JMS que es situarà en el centre de la comunicació entre aplicacions: això permet que es pugui canviar de servidor JMS sense modificar el codi de les aplicacions, ja que el programador usarà la interfície definida per *Sun Microsystems*. Com a servidors de JMS podem destacar:

Programari lliure:

- JBoss
- ActiveMQ
- Fuse Message Broker
- JORAM
- Open JMS

Programari de propietari:

- TIBCO EMS
- WebSphere MQ
- WebSphere Application Server
- Bea Weblogic

La majoria de servidors JMS inclouen consoles i APIs d'administració per poder realitzar la gestió dels mateixos:

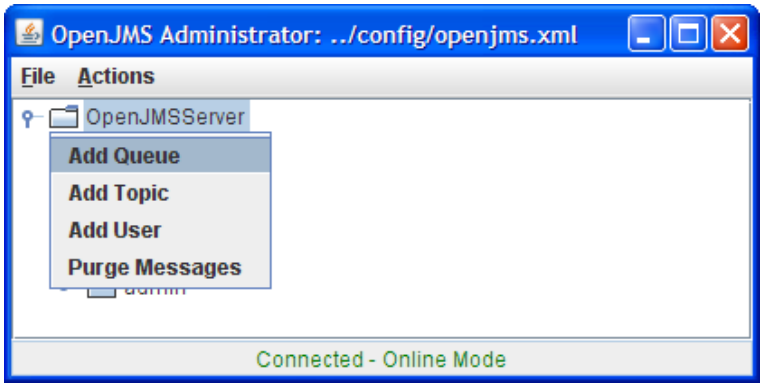


Figura 1: Consola d'administració de OPENJMS

No obstant, en la pràctica acostumen a ser aplicacions CLI (*command-line interface*) o GUI (*graphical user interface*) enfocades a l'administració del servidor de missatgeria i no als rols que efectuen els programadors o els equips de proves, centrats en les aplicacions que utilitzen el servidor per comunicar-se i no en l'administració del mateix. Per tant, es podrien resumir les necessitats d'aquests en:

- Control dels missatges entrants i sortints al servidor JMS, per veure la càrrega que hi ha al sistema.
- Control dels missatges acumulats (no consumits) al servidor.
- Enviament de missatges a una destinació (tòpics o cues), de tal forma que l'usuari ràpidament pot realitzar proves amb els seus desenvolupaments generant un missatge en una destinació concreta.
- Eliminació de missatges no consumits en una destinació (tòpics o cues).
- Interfície gràfica intuïtiva i fàcil d'utilitzar.
- Gestió de servidors favorits, per tal de no introduir totes les dades requerides per la connexió, cada vegada que es vulgui monitoritzar un servidor JMS.

1.2 OBJECTIUS DEL PROJECTE

L'objectiu més rellevant a assolir amb el projecte és el disseny i la implementació d'una eina fàcilment ampliable per la monitorització de servidors JMS, tal que el model inicial de la primera versió contingui les següents funcionalitats:

- **GUI**: la disponibilitat d'una interfície gràfica per la representació de la informació fa més comprensible les dades obtingudes i facilita l'ús a l'usuari.

- **Funcionalitat mínima d'administració:** malgrat l'objectiu no és la gestió pròpia del servidor, el monitor ha de contenir unes funcionalitat mínimes que poden ser utilitzades pels programadors o provadors. Aquestes poden ser la creació i eliminació de destinacions (cues o tòpics) i l'eliminació dels missatges que persisteixen al servidor.

- **Independència de plataforma:** l'usuari pot treballar amb diferents sistemes operatius, per tan és necessari que l' aplicació pugui córrer sota qualsevol plataforma.

- **Aplicació enfocada al tràfic de missatges:** la disposició dels elements i objectes gràfics, la informació recollida i mostrada per l'aplicació, i les funcionalitats per interactuar amb les cues i tòpics estan enfocades al tràfic de missatges entrants i sortints que es registra en el servidor.

- **Independència de servidor JMS:** no té sentit fer un monitor per un servidor en concret, ja que l'aplicació ha d'aprofitar una de les avantatges de la API JMS, com és la independència del codi amb la implementació proporcionada pel servidor amb el que s'estigui treballant.

Tots aquest objectius descrits anteriorment són de caràcter funcional i resumeixen els serveis que ha de proporcionar una aplicació de monitorització de servidors JMS. No obstant, una de les fites més importants en el projecte és la independència del monitor amb els proveïdors JMS, de tal manera que tingui una amplia compatibilitat amb diferents servidors de missatgeria JMS. Per tant, el disseny de l'aplicació a implementar ha de contemplar futures actualitzacions d' implementacions de servidors JMS, minimitzant el cost del manteniment i ampliacions sobre l'aplicació a desenvolupar.

Així doncs, el monitor de servidors JMS, ha d'esdevenir una solució ràpida i pràctica pels programadors i equips de proves que desitgen un monitor amb funcionalitats bàsiques, enfocat al tràfic de missatges i al contingut dels mateixos, ampliable en funció de les seves necessitats i compatible amb diferents proveïdors de missatgeria JMS.

1.3 ESTAT DE L'ART

A dia d'avui, no existeixen gaires solucions que acompleixin amb totalitat amb totes les necessitats descrites anteriorment; si que és cert que els proveïdors inclouen en les seves implementacions consoles d'administració i API, però naturalment el perfil del destinatari d'aquestes eines és el propi administrador del servidor JMS, quan el monitor que es pretén ha d'incloure a més eines com per exemple, l'enviament de missatges i la lectura de missatges enregistrats al servidor. Una de les poques aplicacions que acompleixen d'una forma excel·lent els propòsits descrits en els anteriors apartats és *HermesJMS*.

HermesJMS és un consola extensible que ajuda a l'usuari a interactuar amb els proveïdors JMS navegant per coles o tòpics, que permet enviar missatges, copiar-los i eliminar-los de les destinacions escollides. Multiservidor, s'integra perfectament amb JNDI[2] i permet gestionar les connexions a servidors JMS de diferents proveïdors.

ActiveMQ	Oracle
ArjunaMQ	Pramati
EMS	SAP
FioranoMQ	SeeBeyond ICAN
HornetQ	SeeBeyond JCAPS
JbossMQ	SonicMQ
Jboss Messaging	WeblogicMQ
JORAM	WebMethods
OpenJMS	WebSphereMQ

Taula 1: proveïdors suportats per HermesJMS

2 **JNDI:** API per a serveis de directori que permet als clients descobrir i buscar objectes a través d'un nom, sent independent de la implementació subjacent.

Cost:

HermesJMS és una aplicació de programari lliure amb llicencia *Apache versió 2*[3], això representa un cost inexistent pels drets del programari, fet que fa d'Hermes una alternativa molt vàlida per qualsevol tasca de monitorització de servidors JMS.

Característiques principals:

La interfície gràfica es basa en una finestra principal (on es mostrarà la informació de les destinacions seleccionades) i un marc amb totes les sessions configurades. Per tant, no hi ha un resum global de l'estat del servidor, si no que, a mesura que es selecciona una destinació, es mostren les dades relacionades amb aquesta.

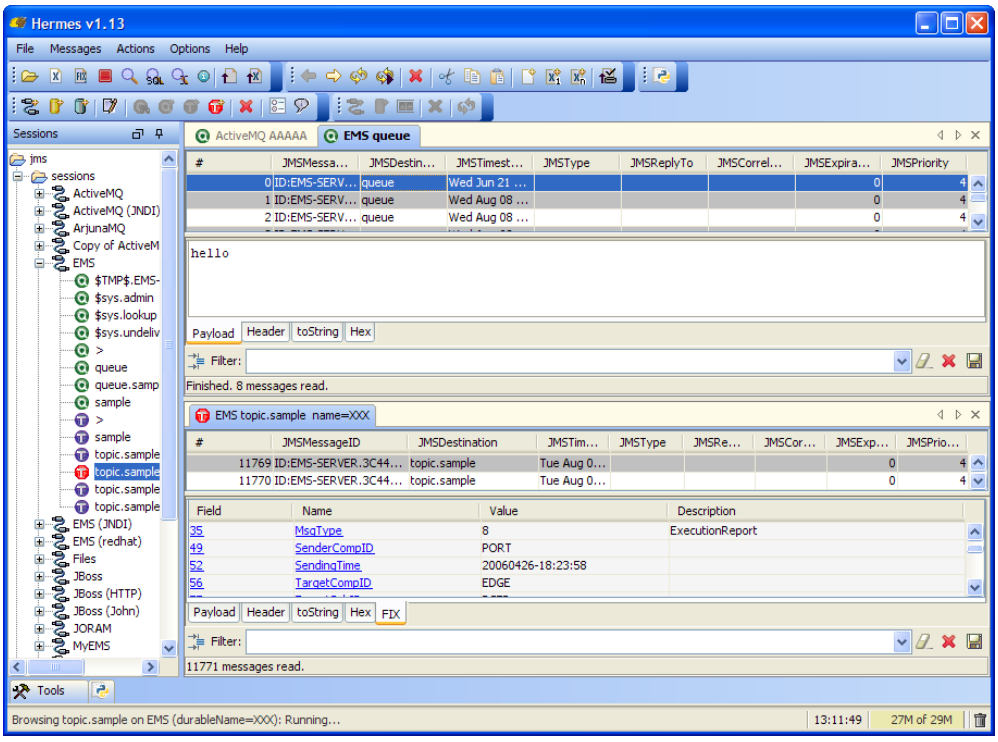


Figura 2: Hermes JMS

3 **Apache license v2.0:** és una llicència de programari lliure creada per l'Apache Software Foundation (ASF). Permet a l'usuari del programari plena llibertat d'ús per a qualsevol propòsit, distribuir-lo, modificar-lo, i distribuir versions modificades d'aquest programari. Més informació a http://www.apache.org/licenses/LICENSE-2.0

És possible accedir al contingut dels missatges sense esborrar-los en el servidor, a més de filtrar-los i exportar-los a fitxers XML[4]. Juntament amb la possibilitat d'enviar i esborrar missatges, Hermes JMS permet cercar cadenes de text en:

- Les capçaleres dels missatges
- Text en missatges de tipus *TextMessage*
- El resultat de trucar al mètode *toString* de qualsevol objecte en un *ObjectMessage.*
- Totes les claus i valors de un missatge *MapMessage*.

Una altra característica interessant és el conjunt de funcionalitats sobre emmagatzematge de missatges rebuts. D'una banda, permet l'intercanvi de missatges entre destinacions, és a dir, l'usuari pot moure missatges d'una cua a un tòpic fent un *drag&drop* del missatge en la interfície gràfica. D'altra, implementa funcionalitats d'enregistrament de missatges rebuts en les destinacions a un lloc extern en el servidor (disc dur, base de dades). Tot i això cal dir, que aquesta funcionalitat és experimental.

Però sense dubte, la gran avantatge de l'aplicació és el seu suport a una gran quantitat de proveïdors. A més d'obtenir dades concretes de les destinacions mitjançant la interfície de l'estàndard JMS, permet la possibilitat d'incloure els connectors (normalment llibreries o API) que els proveïdors faciliten en les seves implementacions, per tal d'obtenir dades i realitzar funcions especifiques per aquell servidor JMS en concret.

4 **XML** o *Extensible Markup Language* és un metallenguatge extensible d'etiquetes desenvolupat pel *World Wide Web Consortium*. És una manera de definir llenguatges per diferents necessitats. Més informació a http://www.w3.org/XML/

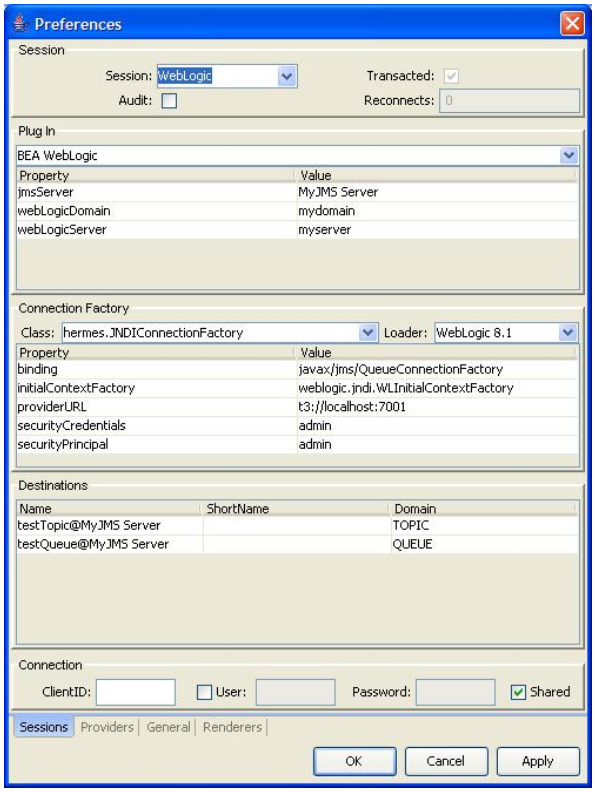


Figura 3: Configuració de connectors de proveïdors a Hermes JMS

Així doncs, *Hermes JMS* és una aplicació que permet realitzar tasques d'administració sobre un gran nombre de servidors de missatgeria Java, on s'inclouen eines i funcionalitats molt interessants sobre els missatges i les destinacions amb un cost nul. No obstant, encara que aquesta alternativa acompleixi amb les funcionalitats bàsiques del propòsit del projecte, el disseny de la interfície fa que el seu ús pels propòsits dels usuaris potencials que s'han descrit anteriorment no sigui gaire efectiu; la navegació per destinacions fa que sigui difícil tenir una visió global de les cues i tòpics. A més, és una aplicació força gran i més complicada de configurar i mantenir, ja que adjunta gestions pròpies d'administració de servidors, funcions que no són necessàries a priori per les tasques dels usuaris als que va destinat el desenvolupament a realitzar en aquest projecte.

Altra alternativa és la utilització de les consoles i eines que proporciona el proveïdor. Aquestes solen englobar totes les funcions relatives a l'administració del servidor; creació de cues,

tòpics, factories, gestió d'usuaris, estat del servidor i estadístiques generals de missatgeria, entre d'altres.

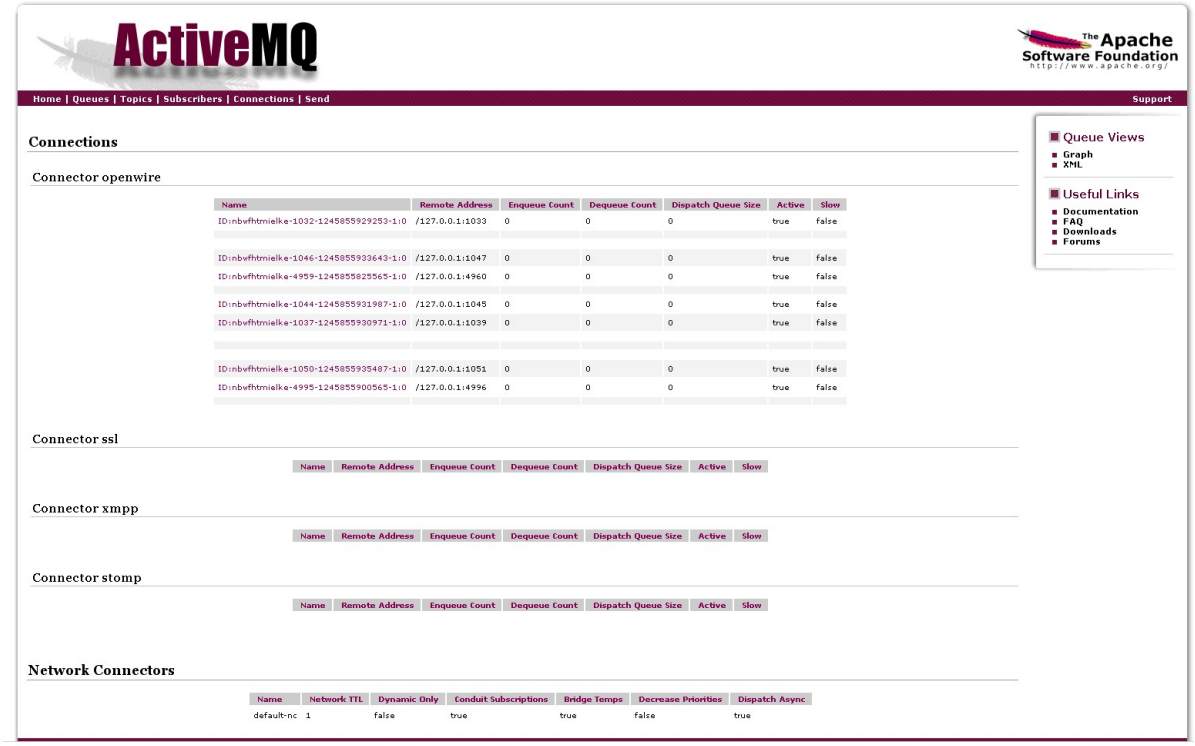


Figura 4: Consola d'administració de Active MQ

No obstant, pels desenvolupadors i equips de proves, no és una alternativa prou efectiva; no solament es vol monitoritzar el servidor, es vol accedir i manipular les destinacions realitzant accions sobre aquestes com per exemple l'enviament de missatges o obtenció del contingut d'aquests, tot en una interfície dissenyada envers la missatgeria i les destinacions en comptes de l'administració del servidor.

1.4 MOTIVACIONS

Actualment, no hi ha gaires aplicacions de monitorització de servidors JMS extensibles a diferents proveïdors, ja que normalment aquests inclouen en les seves implementacions petites aplicacions i APIs d'administració per aquestes tasques. Malgrat tot, de l'experiència laboral pròpia amb servidors JMS, sorgeix la necessitat de disposar d'una eina única per tal de realitzar proves amb desenvolupaments que empren aquesta tecnologia; la lectura de missatges sense consumir-los en el servidor, l'enviament de missatges sobre una destinació escoltada per un client en desenvolupament, el comportament de les aplicacions davant la càrrega massiva de missatges... són tasques rutinàries per programadors i equip de proves.

Així doncs, el monitor de servidors JMS neix com un projecte personal impulsat per una experiència laboral, per l'aprenentatge de tecnologies i com una eina ampliable i gratuïta per tots aquells usuaris que vulguin monitoritzar els seus servidors JMS fàcilment, oferint en un inici funcionalitats bàsiques i una estructura escalable.

1.5 ESTRUCTURA DE LA MEMÒRIA

En aquest primer capítol s'ha realitzat una breu introducció a l'aplicació que es vol desenvolupar, explicant els objectius més importants i destacables juntament amb exemples d'aplicacions existents que desenvolupen tasques semblants a les proposades pel projecte exposat. A més, es comenten les motivacions personals i el per què del monitor de servidors JMS.

En el segon capítol s'analitza la viabilitat tècnica i es realitza una planificació amb les fites a acomplir en el desenvolupament del projecte. Es concretaran els punts i objectius que es pretenen assolir en aquesta primera versió de l'aplicació.

En el tercer capítol s'exposen els fonaments teòrics necessaris per entendre el present projecte, fent un breu resum del que s'entén per un programari d'intermediari o *middleware* i els diferents tipus que es poden trobar, tot fent un major èmfasi en la tecnologia a la qual aquest projecte vol facilitar una eina, la *Java Message Service*.

En el quart capítol es defineixen de manera més precisa les característiques del projecte i es relacionen els diferents requeriments que el conformaran, previs al seu desenvolupament.

En el cinquè i sisè capítols es descriu de manera detallada les diferents fases del procés de disseny i implementació de l'aplicació, incidint en aquelles parts que han suposat un major grau de dificultat tècnica. Es comenten a més aquelles correccions més significatives detectades durant les proves realitzades.

El capítol setè resumeix les conclusions extretes en la realització del projecte i s'analitzen els resultats obtinguts. Es descriuen les línies del projecte que queden obertes i es mostra una bibliografia completa dels recursos que s'han consultat a l'hora de desenvolupar el projecte.

CAPÍTOL 2 Estudi de viabilitat

2.1 INTRODUCCIÓ

En aquest capítol es descriuran de manera més precisa els objectius i requeriments del projecte a desenvolupar per poder avaluar les diferents solucions possibles i determinar si es tracta d'un projecte viable.

Una vegada s'hagin analitzat les diferents propostes s'escollirà, sota un criteri argumentat, quina d'aquestes acompleix i satisfà els objectius i requeriments del projecte en els terminis proposats. A més s'estableixen certes limitacions en el desenvolupament de les funcionalitats, els recursos disponibles, analitzant els riscos futurs i amb una planificació per minimitzar les dificultats que poden aparèixer durant el cicle de desenvolupament del projecte.

S'estudiarà el disseny i l' implementació d'una solució amb interfície gràfica que mostri de forma ràpida i visual l'estat de les destinacions creades en un servidor de missatgeria JMS, que inclourà utilitats per enviar, esborrar i mirar el contingut de missatges. A més, ha de ser fàcil de mantenir i ampliar, ja que l'aplicació pretén ser una solució única pel monitorizatge de qualsevol servidor JMS, destinada a perfils de programadors, equip de proves i suport de aplicacions.

2.1.1 Avantatges de l'aplicació a implementar

- **Simplificació de l' informació de control:** facilita l'administració i supervisió d'un servidor mostrant d'una manera clara els missatges que hi ha en una cua o tòpic, a més d'estadístiques (sempre que el proveïdors JMS tingui disponible aquesta funcionalitat).

- **Seguiment i evolució:** les gràfiques generades sobre l'activitat del servidor ajuden a tenir un registre històric a l'usuari sobre el tràfic de missatges en el servidor. A més, és possible l'exportació d'aquestes a altres formats, facilitant la incorporació de les gràfiques en documents o informes.

- **Independència de sistema operatiu:** l'aplicació pot córrer sota diferents sistemes operatius sense modificar el codi o utilitzar altres tipus de llibreries gràcies a la tecnologia Java.

- **Control del tràfic de missatges:** la interfície està disposada i dissenyada per veure el tràfic de missatges entrants i sortints en les destinacions de forma que, amb un simple cop d'ull al monitor sense cap interacció prèvia de l'usuari, es pot saber si aquest a arribat al seu destí, si ha sigut consumit i el seu contingut.

- **Eina multiservidor:** El monitor està pensat per poder fer les seves tasques sobre qualsevol monitor JMS. Així doncs, tant el disseny com la implementació contemplen la possibilitat d'ampliar fàcilment l'aplicació a mesura que es treballin amb diferents servidors JMS i es requereixi d'una actualització del monitor, ja que, la implementació de la comunicació amb el servidor és independent de l'aplicació gràfica.

• **Facilitat d'ús:** la interfície gràfica, fa que la interacció amb el monitor per part de l'usuari sigui menor en comparació amb una interfície de línia de comandes.

2.1.2 Inconvenients de l'aplicació a implementar

- **La monitorització consumeix recursos de la xarxa:** la comunicació amb el servidor JMS requereix de la pròpia missatgeria JMS per obtenir les dades rellevants a la seva activitat. Aquest és un fet decisiu a l'hora de realitzar el disseny del monitor, ja que s'ha de minimitzar l'impacte dels missatges de monitorització en el sistema de missatgeria Java, per tal d'afectar el menys possible al funcionament normal del servidor.

- **Aplicació *stand-alone*:** El monitor s'executa en totes aquelles màquines on treballen els usuaris que volen utilitzar-lo; no és una solució centralitzada que corre sobre un ordinador accessible a tots els usuaris i atén a les peticions d'aquests per accedir a la informació com si es tractes d'una *intranet*. És una eina per ús local, fet que provoca que per cada connexió per realitzar la connexió amb el servidor JMS, la quantitat de missatges de monitorització vagi incrementant. Per això, és un tema crític que el disseny ha de resoldre, minimitzant la quantitat de missatges obtinguts del servidor.

2.2 OBJECTE

2.2.1 Descripció de la situació a tractar

Es parteix de la necessitat de monitoritzar el tràfic de missatgeria en un servidor, per tal de respondre a les necessitats que es donen en tot projecte on s'empra la tecnologia JMS, de tal forma que els programadors, provadors i el suport tècnic de sistemes, tinguin una eina pràctica i fàcil d'utilitzar que els faci estalviar temps davant les consoles d'administració, no enfocades al tràfic i a la missatgeria i sí a tasques pròpies de gestió del propi servidor.

Així doncs, el monitor de servidors JMS ha d'aportar, a més de la pròpia monitorització, un conjunt de facilitats i funcionalitats pels rols esmentats anteriorment que li donin un valor afegit i permeti agilitzar tasques com:

- **Determinació de problemes de connexió de clients al servidor:** En qualsevol instant, es poden veure el nombre de consumidors que te una destinació.

- **Anàlisi del comportament del servidor davant sobrecarrega de missatges:** Una gran quantitat de missatges per segon pot fer que els recursos demandats pel servidor acabin per enfonsar la màquina i alenteixi tot el sistema de missatgeria en general. Aquesta situació no només afecta al servidor JMS, si no a tots els clients connectats a ell, ja que els missatges s'aniran acumulant al servidor per ser consumits posteriorment, fet provocat per l'embut creat pels clients destinataris al no poder processar els missatges pendents amb el mateix ritme que els que es generen nous. En conseqüència, incrementa dramàticament la memòria i l'escriptura a disc (en cas de les cues, per les seves propietats de persistència de la informació) i que els clients que han de rebre altres missatges es vegin afectats pel minvament de recursos en el servidor, baixant l'eficiència del sistema (components que necessiten compartir informació i servidor de missatgeria).

- **Proves de components:** La inclusió de funcionalitats, com per exemple l'enviament de missatges, permeten simular el comportament i realitzar proves unitàries i globals dels components que conformen el sistema a desenvolupar.

- **Redacció d'informes i estadístiques:** Incorporant aquestes a una aplicació de monitorizatge, l'usuari pot exportar-les a format gràfic per utilitzar-les amb un processadors de textos o altres aplicacions, a més d'obtenir una ràpida visió de l'estat del servidor JMS.

- **Verificació del contingut dels missatges emmagatzemats al servidor:** És possible que sigui necessari contrastar el contingut dels missatges amb el resultat esperat per un client productor en concret durant una fase de desenvolupament o proves.

2.2.2 Perfil de l'usuari

Aquesta aplicació està orientada a tots aquells perfils que analitzen, desenvolupen i mantenen aplicacions que necessiten de la comunicació entre les mateixes mitjançant la figura *middleware* del servidor JMS. Es distingeixen els següents rols:

- **Programadors:** tots aquells individus que fan tasques de desenvolupament o utilitzen aquests servidors i necessiten comprovar i consultar l'estat en que es troben aquests de manera ràpida.

- **Equip de proves:** Usuaris que fan proves sobre els desenvolupaments implementats per l'equip anterior. Necessiten eines per poder interactuar amb el sistema i comprovar les entrades amb els resultats esperats.

- **Equip de suport del sistema:** Són el punt de connexió del sistema a desenvolupar o desenvolupat amb el client. Gestionen i fan un primer anàlisi sobre incidències en l'ús de l'aplicació, per tant una eina que analitzi els continguts del missatges és útil per determinar les causes del problema. En cas de que sigui una errada en el programari, deriven la responsabilitat en forma d'una solicitud de correcció als programadors.

2.2.3 Objectius

El monitor de servidors JMS que es pretén desenvolupar en el present projecte, ha de ser una eina amigable per l'usuari amb una representació de la informació clara. Des del procés d'instal·lació, passant per la configuració i fins l'ús rutinari del programa, ha de resultar una experiència que faciliti tots els processos d'implantació de l'aplicació.

La idea principal del monitor JMS és la de reflectir l'estat d'un servidor JMS en temps real. Degut a la pròpia filosofia JMS, el monitor ha de ser capaç de suportar la connexió per diferents proveïdors de *Java Message Service*, proporcionant mecanismes que facilitin la integració d'aquests amb el present desenvolupament minimitzant els costos derivats de l'ampliació i manteniment del monitor.

Agilitzar les tasques de l'usuari és sens dubte una de les raons del naixement del projecte; l'enviament de missatges a una destinació per observar com es comporten els components d'un sistema distribuït, la realització de proves de càrrega sobre el servidor de missatgeria, la generació de gràfiques sobre paràmetres monitoritzats i la consulta de contingut de missatges són aspectes necessaris en les activitats realitzades pels usuaris potencials del sistema, anomenats en l'anterior apartat.

Per facilitar la configuració de les connexions als servidors JMS, el propi client gràfic ha de permetre aquesta gestió, fent que l'usuari estalvii temps introduint dades cada vegada que es

vulgui connectar a un servidor JMS, que a la pràctica normalment solen ser una llista definida i poc canviant. La gestió de tipus d'usuari a l'aplicació no és un qüestió a tenir en compte, ja que és l'administrador del servidor JMS qui decideix en funció de la definició d'usuaris i permisos en la pròpia configuració del servidor, fins on pot gestionar l'usuari del monitor.

El monitor JMS, és una aplicació que neix amb una vocació personal d'aprenentatge on la versió desenvolupada en aquest projecte, vol ser el primer esglaó d'una eina al servei de tothom, de cost mínim, facilitant el disseny i el codi a la comunitat d'usuaris i programadors.

2.3 DESCRIPCIÓ DEL SISTEMA

Existeixen diverses tecnologies i llenguatges de programació que poden ser vàlids per cercar una solució al projecte exposat. Cal recordar, que es demana una aplicació simple i de baix cost de manteniment que pugui créixer en funció de les necessitats del client. En el següent apartat es descriuran les diferents plataformes i tecnologies analitzades i es justificarà l'elecció d'aquella que ha sigut escollida. Seguidament s'indicaran els diferents riscos que hi podrien aparèixer durant el desenvolupament i els recursos que s'hi disposaran per a la seva realització.

2.3.1 Anàlisi dels llenguatges de programació

D'una banda tenim llenguatges compilats[5] com C (*paradigma imperatiu*) o C++ (*multiparadigma*). C és molt eficient però té un cost de desenvolupament més gran, ja que es requereix més temps per realitzar una funcionalitat que altres llenguatges de més alt nivell. C++ permet un major nivell d'abstracció i es pot definir com *multiparadigma*, ja que afegeix a C eines per la manipulació d'objectes però al mateix cop és possible seguir un model de programació estructurat. Tot i que els dos són dels més eficients, al ésser llenguatges compilats, el cost de la portabilitat és major: cal modificar el codi i utilitzar diferents llibreries, a més de compilar l'aplicació de nou en funció de la plataforma on s'executarà el programa.

Els llenguatges interpretats com *Perl* o *Python* gaudeixen d'una major possibilitat de ser portables i fan més ràpida la implementació del codi pel seu alt nivell d'abstracció. *Perl* permet la comunicació entre ordinadors sota protocols TCP/IP; implementa fils d'execució o *threads*; i a més té la opció de suportar l' estil de programació orientat a objectes, fet que facilita el disseny i codificació de qualsevol tipus de programa. *Python* suporta els paradigmes de programació orientada a objectes, programació estructurada i programació funcional, a més de ser un

5 En els **llenguatges compilats**, el compilador tradueix el codi d'alt nivell a codi màquina o de baix nivell per a que pugui ser executada, generant així un arxiu executable per a una plataforma en concret.

llenguatge fàcil d'aprendre i de disposar d'una sintaxi molt llegible. No obstant, en diferencia amb els compilats, un llenguatge interpretat necessita un intèrpret per executar el codi; això fa que en la majoria dels casos siguin programes més lents. Són llenguatges molt utilitzats per fer prototips i complements o connectors per altres programes.

Pel desenvolupament del projecte s'ha escollit Java, llenguatge orientat a objectes i amb una alta portabilitat a nombroses plataformes: el mateix codi pot córrer sobre diferents màquines. El codi Java és compilat primer generant un codi *bytecode*[6] per a que la *Java Virtual Machine*[7] que estarà escrita en codi natiu a la plataforma en concret l'executi. A diferència d'altres, la màquina virtual de Java utilitza un compilador en temps d'execució fet que millora el rendiment de sistemes de programació que utilitzen *bytecode*. Cal dir, que no és tan ràpid com un executable natiu per a una determinada plataforma, però la seva amplia API, la portabilitat i la gestió automatitzada de la memòria; fet que redueix el temps d'implementació i elimina errors clàssics en la programació com són les fuites de memòria o *memory leaks* provocats per la mala gestió en l'alliberació de la memòria. Totes aquestes avantatges el fan un bon candidat pel desenvolupament del projecte, però sens dubte el motiu principal de la seva elecció és la tecnologia que es vol monitoritzar; les API's d'administració i accés a servidors JMS estan desenvolupades principalment en Java.

És important comentar, que la distinció realitzada en els anteriors paràgrafs entre interpretat i compilat, és purament pràctica i no per propietats inherents al propi llenguatge. En teoria, qualsevol llenguatge de programació pot ésser compilat o interpretat com per exemple *Lisp*, *Basic* o el propi *Python,* que disposen de compiladors i intèrprets.

6 El ***bytecode*** es situa en mig del codi màquina i del d'alt nivell. Rep aquest nom ja que normalment la majoria de codis d'operació té la longitud d'un byte.

7 La ***Java Virtual Machine*** o màquina virtual de Java, és un executable per a una determinada plataforma que interpreta i executa les instruccions en Java *bytecode*, codi resultant del compilador de Java.

2.3.2 Anàlisi de proveïdors JMS

A l'actualitat existeixen un gran nombre de proveïdors de tecnologia JMS. Com que el projecte és independent de proveïdors i es pretén reduir costos, es descartarà el programari privatiu i de pagament. Així doncs s'escolliran dos proveïdors que no afectin als costos del present projecte i que al mateix temps siguin productes emprats per diferents desenvolupaments i projectes que utilitzin la tecnologia JMS.

Fuse Message Broker és un gestor de missatgeria JMS basat en *Apache Active MQ* alliberat per *Fuse Open Souce Community*. Amb una llicència *Apache v.2.0*, és un producte provat i certificat amb un equip de suport darrera, que disposa d'una notable capacitat per gestionar gran volum de dades i amb un dels millors rendiments dintre del proveïdors de missatgeria JMS. A més és compatible amb JMS 1.1 i altres tecnologies i protocols com poden ser com JDBC, JCA, i EJB, AJAX, REST, HTTP, TCP, SSL, NIO i UDP. Les característiques principals d'aquest proveïdors són:

FUSE MESSAGE BROKER	
CARACTERISTIQUES	**AVANTATGES**
Basat en estàndards – compatible amb JMS, J2EE, JNDI,AJAX, REST, HTTP	**Facilitat de desenvolupament**– connectivitat sense fissures amb actius nous i existents.
Publicador i subscriptor / missatgeria punt a punt – permet *broadcasting* i/o missatgeria única.	**Plataforma completa de missatgeria** – compatible amb durables i connexions d'alta disponibilitat.
SEDA, compressió, *prefetch*, JMS *streams*	**Millor *performance*** – capçar de gestionar grans carregues de missatges i volum de dades.
Alta disponibilitat – clúster, capacitat per recuperar dades davant caigudes.	**No pèrdua de dades** – resistent a les fallides de xarxa o de sistemes.
Autentificació / autorització – compatible amb aplicacions pròpies i solucions de tercers.	**Fàcil integració** – aprofita les inversions existents en seguretat.

Taula 2: característiques i avantatges de Fuse Message Broker

D'altra banda, o*penJMS* és una implementació de programari lliure de la API *Java Message Service 1.1*. Un dels beneficis de *openJMS* és que és un proveïdor neutral. A causa de les especificacions de JMS Java no s'especifica un protocol de connexió, l'aplicació de cada proveïdor de JMS és diferent i no és interoperable amb altres. Les implementacions de JMS normalment són alliberades com a part d'un proveïdor de servidor d'aplicacions, això fa l'aplicació d'un proveïdor de JMS específica per aquest servidor d'aplicacions. *OpenJMS*, al igual que FUSE, no depèn de cap servidor d'aplicacions i per tant pot ser una interfície comú entre els usuaris de diferents proveïdors. Els usuaris de servidors d'aplicacions diferents poden posar-se d'acord per utilitzar l'aplicació *OpenJMS* comuns per a la interoperabilitat de les seves capes. Destaquen les següents funcionalitats:

OPEN JMS	
CARACTERISTIQUES PRINCIPALS	**AVANTATGES**
Publicador i subscriptor / missatgeria punt a punt – permet *broadcasting* i/o missatgeria única.	**Plataforma completa de missatgeria** – compatible amb durables i connexions d'alta disponibilitat.
Garantia d'entrega de missatges	**Certificat d'entrega -** Els clients productors s'assabenten si el consumidor a rebut el missatge correctament.
Persistència amb JDBC	**Compatibilitat amb bases de dades –** Permet emmagatzemar les dades del missatges a una base de dades per tal de recuper la informació en cas de fallida del servidor.
Autentificació	**Seguretat d'accés –** Permet la gestió d'usuaris i l'accés al servidor de missatgeria.
Detecció automàtica de desconnexions de clients	**Millora de rendiment –** El servidor detecta que aquell client no es troba disponible i l'elimina de la seva llista de connexions obertes.
Compatible amb TCP, RMI, HTTP i SSL	**Millora integració** - Afavoreix la integració i la capacitat de desenvolupament amb altres aplicacions.

Taula 3: característiques i avantatges de OpenJMS

Tots els proveïdors anomenats anteriorment poden ser executats en plataformes *Linux* i *Windows* i suposen un cost nul en llicències pels desenvolupaments que optin per un d'ells com

intermediari de comunicacions JMS.

2.3.3 Anàlisi d'entorns de desenvolupament integrats

Un entorn de desenvolupament integrat o IDE, facilita al programador la seva tasca oferint múltiples funcionalitats de les que es poden destacar editor de text, compilador, control de versions, casos de test, i depurador. *Eclipse* i *Netbeans* són els IDE més coneguts pel desenvolupament en Java. Els dos disposen de les eines esmentades anteriorment i a més accepten connectors per ampliar l'entorn, des de suport per altres llenguatges (com *PHP, C, C++, Ruby, Perl*) fins editors de UML[8]. En el projecte s'utilitzarà Eclipse, que consta d'un suport molt ampli de connectors i d'un editor de text excel·lent.

2.3.4 Anàlisi de llibreries gràfiques

A fi de poder implementar les gràfiques que generarà l'aplicació sobre les estadístiques monitoritzades, s'opta per escollir una llibreria externa que permeti que el desenvolupament d'aquesta funcionalitat sigui més ràpid.

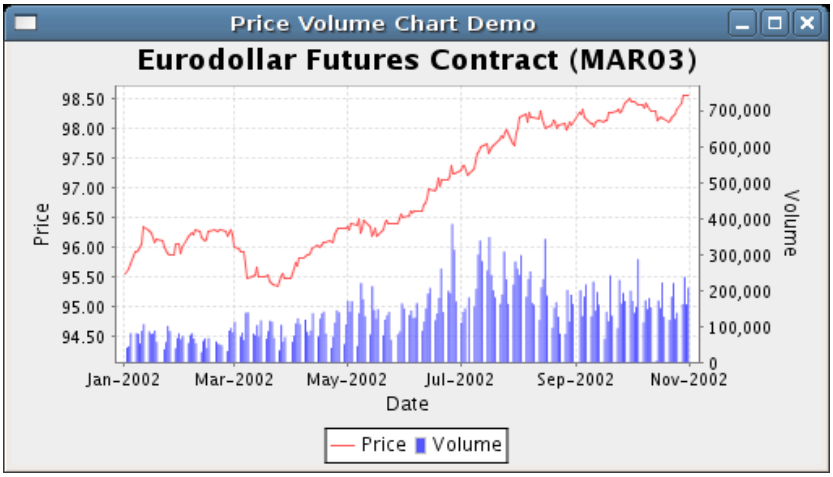


Figura 5: gràfica generada amb la llibreria JfreeCharts

8**UML** o *unified modeling language* és un llenguatge de modelatge i definició de sistemes de programari. Més informació en http://www.uml.org/.

JfreeCharts, és una llibreria per Java amb llicència LGPL[9], amb una API simple i fàcil d'entendre. Permet realitzar una gran quantitat de gràfiques, amb eines incrustades com l'ampliació o disminució de la gràfica sobre el plànol i l'exportació d'aquestes a .PNG.

Elegant Jcharts és una altra alternativa per la generació de gràfiques en Java. Capaç de generar gran quantitat de gràfiques amb efectes 2D i 3D, inclou també la capacitat *d'scrolling* que tenia *JfreeCharts* amb una qualitat gràfica superior. No obstant, aquesta llibreria requereix d'un cost de llicència, fet que situa a l'anterior com a la candidata pel present projecte.

2.3.5 Avaluació de riscos

Com s'ha comentat a apartats anteriors, l'aplicació ha de ser executada per cada usuari, no es tracta d'un monitor centralitzat a una màquina a la qual es pot tenir accés, si no que és un client que visualitza l'estat de les cues, tòpics i els missatges que existeixen a un servidor JMS. Així doncs, l'ús massiu i distribuït d'aquesta eina pot provocar un tràfic elevat de missatges a la xarxa. A més, també s'ha de tenir en compte que són peticions que ataquen contra el servidor JMS, fet que influeix en el rendiment d'aquest, ja que ha de generar la resposta i respondre amb la dada sol·licitada.

D'altra banda, la gran varietat de proveïdors JMS fa d'inici que no es puguin desenvolupar un gran nombre de les interfícies que donen accés als diferents servidors amb el nombre d'hores valorades pel present projecte, fet que implicarà que els futurs usuaris que utilitzin diferents proveïdors hagin d'implementar la interfície. No obstant s'entregaran llibreries que implementaran l'accés de dos proveïdors.

Cal remarcar la falta d'experiència prèvia i ús de tecnologies pel desenvolupament del

9 **GNU Lesser General Public License** és una llicència *open source* que permet l'ús del programari o llibreries a altres aplicacions de propietari. Més informació en: http://www.gnu.org/licenses/lgpl.html

projecte, fet que pot influir en el acompliment de les dades previstes per l'entrega. A més, es tracta d'un projecte amb un cicle de salt d'aigua, amb el que qualsevol demora en una fase anterior endarrereix la següent.

2.3.6 Recursos

	RECURSOS HUMANS
Rol	• Analista • Tècnic programador • Equip de proves

Taula 4: recursos humans

	PROGRAMARI
Servidor monitoritzat	• Sistemes Operatius: Ubuntu 9.04 • Java Runtime Environment 1.6
Client per la monitorització	• Sistema Operatiu: Ubuntu 9.04 / Windows XP • Java Runtime Environment 1.6
Entorns de programació	• Diagrames UML: DIA • Eclipse
Generació de documentació	• OpenOffice 3.0
Llibreries i aplicacions de tercers	• JDK 1.6 • OpenJMS • FUSE • JFreeCharts
Gestió del projecte	• Planificació: Microsoft Project

Taula 5: programari disponible

	MAQUINARI
Recursos mínims clients*	• Memòria: 256 MB* • Processador: AMD *Sempron* 2800+ 1600 Mhz / mem. cau L2: 256KB • Disc dur: 5 GB • Unitat de reproducció de DVD • Monitor SVGA • Tarja de xarxa • Teclat i ratolí ** la memòria mínima ve imposada en la majoria dels casos pels requisits mínims del sistema operatiu, ja que el monitor no té un consum molt elevat. Per Windows Vista, la memòria mínima recomanable és de 1 GB.*
Recursos mínims servidor	• Memòria: 2048 MB • Processador: Intel Pentium Dual Core E6300 2800 Mhz / mem. cau L2: 2048 KB • Unitat de reproducció i enregistrament de DVD • Disc dur: 10 GB • Tarja de xarxa

Taula 6: maquinari disponible

2.4 ORGANITZACIÓ DEL PROJECTE

En primera instància, serà necessària una cerca de requeriments i funcionalitats bàsiques que tot sistema de monitorització ha de complir. Observant els productes existents al mercat es podrà acotar els objectius d'aquest projecte amb els requeriments mínims.

Després d'una primera fase de documentació inicial i estudi de la viabilitat, es perfilaran els requeriments funcionals i no funcionals, generant un diagrama de casos d'ús. També es realitzarà la cerca de documentació i l'estudi de les diferents tecnologies disponibles pel desenvolupament del projecte, tot instal·lant les eines i l'entorn de desenvolupament.

Una vegada establerts els requeriments funcionals i no funcionals i l'entorn tecnològic amb que es desenvoluparà el projecte, es realitzarà el disseny dels mòduls que conformen l'aplicació per codificar i implementar el codi. Posteriorment es farà una fase de proves, per poder fer correccions i millores funcionals. La memòria es redactarà al mateix temps que es desenvolupen les distintes fases del projecte.

Per a la documentació del programari i les tecnologies s'estableixen les següents vies per captar informació; documentació oficial de les API, els entorns de programació i llibres sobre les tecnologies emprades, tutorials i experiències de programadors (blogs, fòrums) i memòries de projectes anteriors semblants.

2.5 ANÀLISI COST - BENEFICI

2.5.1 Costos recursos materials

De entrada, es disposen de recursos per realitzar el projecte i no caldria l'adquisició de nous. No obstant s'ha de tenir en compte la amortització dels recursos, que s'ha calculat sobre 3 anys ja que el maquinari i el programari solen canviar o renovar en aquest període de temps.

AMORTITZACIÓ DELS COSTOS	
Maquinari	
Compaq CQ60 Laptop Intel Dual Core 3GB RAM 250GB HDD (499€)	5,81 €
AMD 3600 Sempron 2,5GB RAM 250GB HDD Monitor Acer 20" (700€)	8,16 €
Router Comtrend 536+ (50€)	0,58 €
Programari	
Ubuntu 9.04	0,00 €
Windows XP Profesional SP3 (118€)	1,37 €
Windows Vista Home Basic SP1 (210€)	2,45 €
Eclipse IDE	0,00 €
JDK 1.6	0,00 €
JRE 1.6	0,00 €
DIA	0,00 €
Openoffice 3.0	0,00 €
Microsoft Project 2003 (490€)	5,71 €
OpenJMS	0,00 €
FUSE	0,00 €
JfreeChart	0,00 €
TOTAL	**24,08 €**

Taula 7: costos recursos materials

2.5.2 Costos recursos humans

RECURS	€/h
Analista	**18**
Tècnic programador	**10**
Provador	**9**

Taula 8: cost de recursos per hora

RECURS	COST	HORES
Analista	**2.826,00 €**	**157**
Descripció del projecte	54,00 €	3
Estudi d'alternatives	72,00 €	4
Anàlisis de viabilitat	360,00 €	20
Definició necessitats reals	72,00 €	4
Interficie gràfica	432,00 €	24
Gestor de connexions	270,00 €	15
Interficie amb els proveïdors JMS	144,00 €	8
Funcionalitats amb destinacions i missatges	180,00 €	10
Gràfiques	144,00 €	8
Memòria	1.098,00 €	61
Tècnic programador	**1.340,00 €**	**134**
Configuració de l'entorn de desenvolupament	20,00 €	2
Gestor de connexions	200,00 €	20
Interficie amb proveïdors JMS	100,00 €	10
Implementació de llibreria per OpenJMS	140,00 €	14
Implementació de llibreria per ActiveMQ (Fuse)	120,00 €	12
Gràfiques	120,00 €	12
Funcionalitats amb destinacions i missatges	120,00 €	12
Interficie gràfica	240,00 €	24
Correcció d'incidencies	100,00 €	10
Millores disseny / usabilitat	100,00 €	10
Annexos (documentació)	80,00 €	8
Provador	**99,00 €**	**11**
Proves locals	54,00 €	6
Proves finals	45,00 €	5
TOTAL COSTOS RRHH		**4.265,00 €**

Taula 9: costos RRHH

2.5.3 Pressupost

Factura	
Aplicació de monitor de servidors JMS *Instal·lador* *Monitor:* *-Enviament i esborrat de missatges pendents* *-Generació de gràfiques* *-Gestió de connexions* *-Consulta del contingut missatges pendents* *-Visualització d'estadístiques*	6.629,21 €
TOTAL	**6.629,21 €**

Taula 10: pressupost

**el preu final no contempla impostos indirectes*

2.6 PLANIFICACIÓ DEL PROJECTE

2.6.1 Model de desenvolupament i etapes del projecte

El projecte seguirà un model de desenvolupament lineal seqüencial, ja que els requeriments funcionals són establerts des de un principi sense riscos de que siguin modificats en fases posteriors a la d'anàlisi de requeriments. No obstant un dels objectius del projecte és la escalabilitat de la aplicació, per tant, no és descarten nous cicles en un futur, una vegada hagi finalitzades les etapes que es descriuen a continuació:

Tasques	Hores	Recurs
Anàlisi de requeriments	**31**	
Descripció del projecte	3	Analista
Estudi d'alternatives	4	Analista
Anàlisis de viabilitat	20	Analista
Definició necessitats	4	Analista
Implementació	**171**	
Configuració de l'entorn de desenvolupament	2	Tècnic programador
Disseny	**65**	
Interficie gràfica	24	Analista
Gestor de connexions	15	Analista
Interficie amb proveïdors	8	Analista
Funcionalitats amb destinacions i missatges	10	Analista
Gràfiques	8	Analista
Codificació	**104**	
Gestor de connexions	20	Tècnic programador
Interficie amb proveïdors	10	Tècnic programador
Implementació de llibreria OPENJMS	14	Tècnic programador
Implementació de llibreria ActiveMQ (Fuse)	12	Tècnic programador
Gràfiques	12	Tècnic programador
Funcionalitats amb destinacions i missatges	12	Tècnic programador
Interficie gràfica	24	Tècnic programador
Avaluació	**31**	
Proves locals	6	Provador
Correcció d'incidencies	10	Tècnic programador
Millores disseny / usabilitat	10	Tècnic programador
Proves finals	5	Provador
Documentació	**69**	
Memòria	61	Analista
Annexos	8	Tècnic programador
TOTAL	**302**	

Taula 11: recursos i planificació

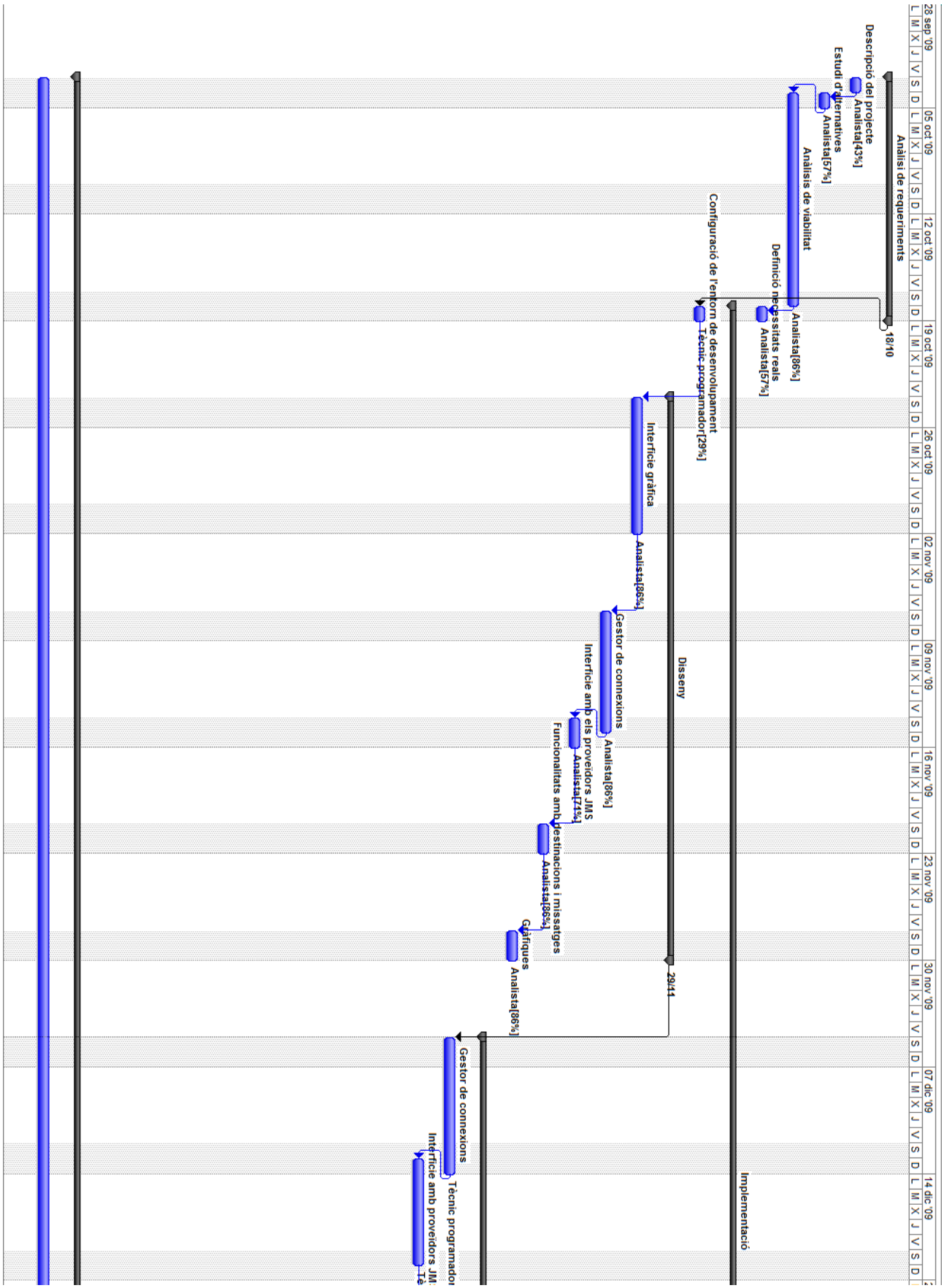
2.6.2 Ús dels recursos

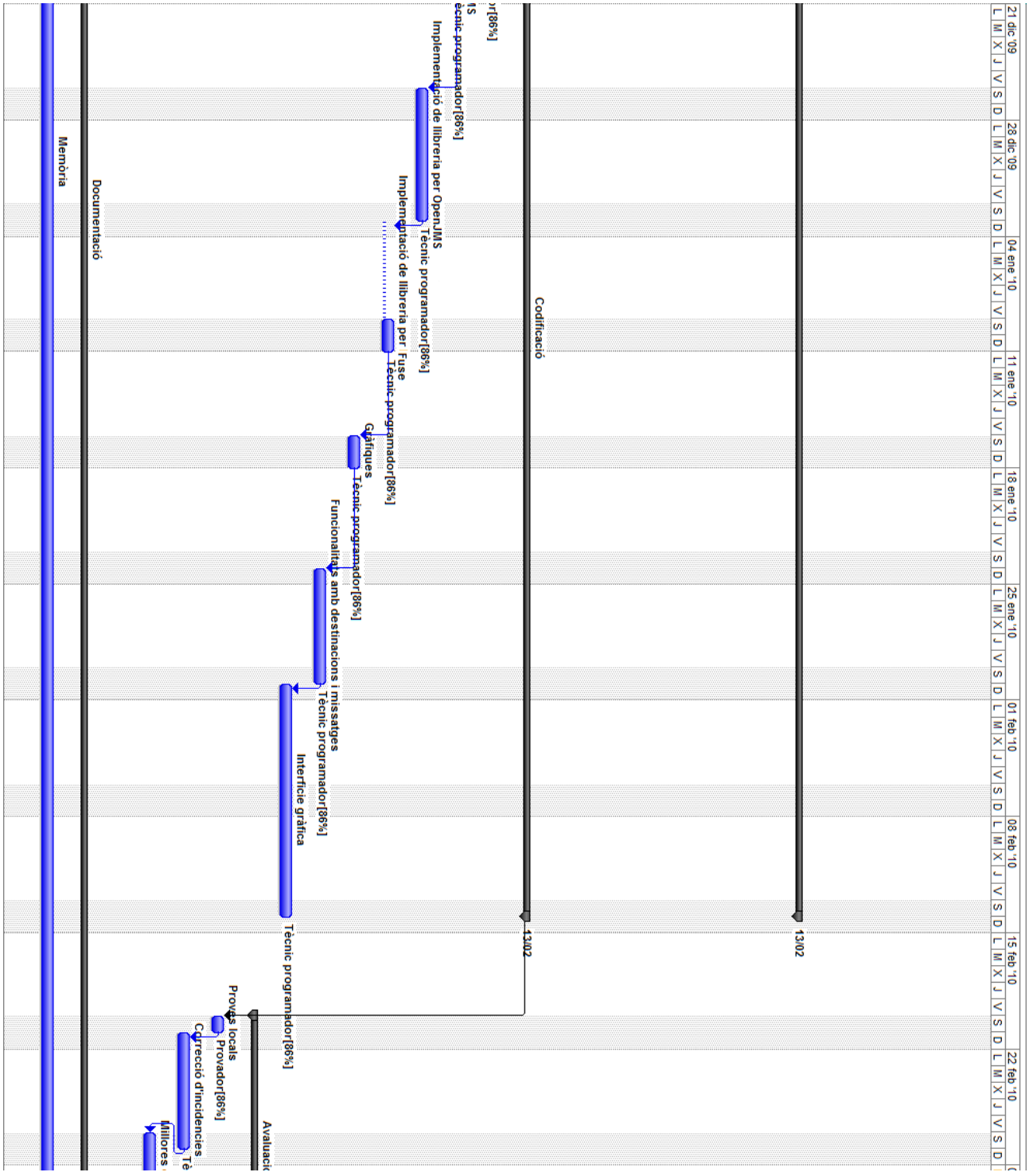
RECURS	HORES
Analista	**157**
Descripció del projecte	3
Estudi d'alternatives	4
Anàlisis de viabilitat	20
Definició necessitats reals	4
Interficie gràfica	24
Gestor de connexions	15
Interficie amb els proveïdors JMS	8
Funcionalitats amb destinacions i missatges	10
Gràfiques	8
Memòria	61
Tècnic programador	**134**
Configuració de l'entorn de desenvolupament	2
Gestor de connexions	20
Interficie amb proveïdors JMS	10
Implementació de llibreria per OPENJMS	14
Implementació de llibreria per ActiveMQ (Fuse)	12
Gràfiques	12
Funcionalitats amb destinacions i missatges	12
Interficie gràfica	24
Correcció d'incidencies	10
Millores disseny / usabilitat	10
Annexos	8
Provador	**11**
Proves locals	6
Proves finals	5

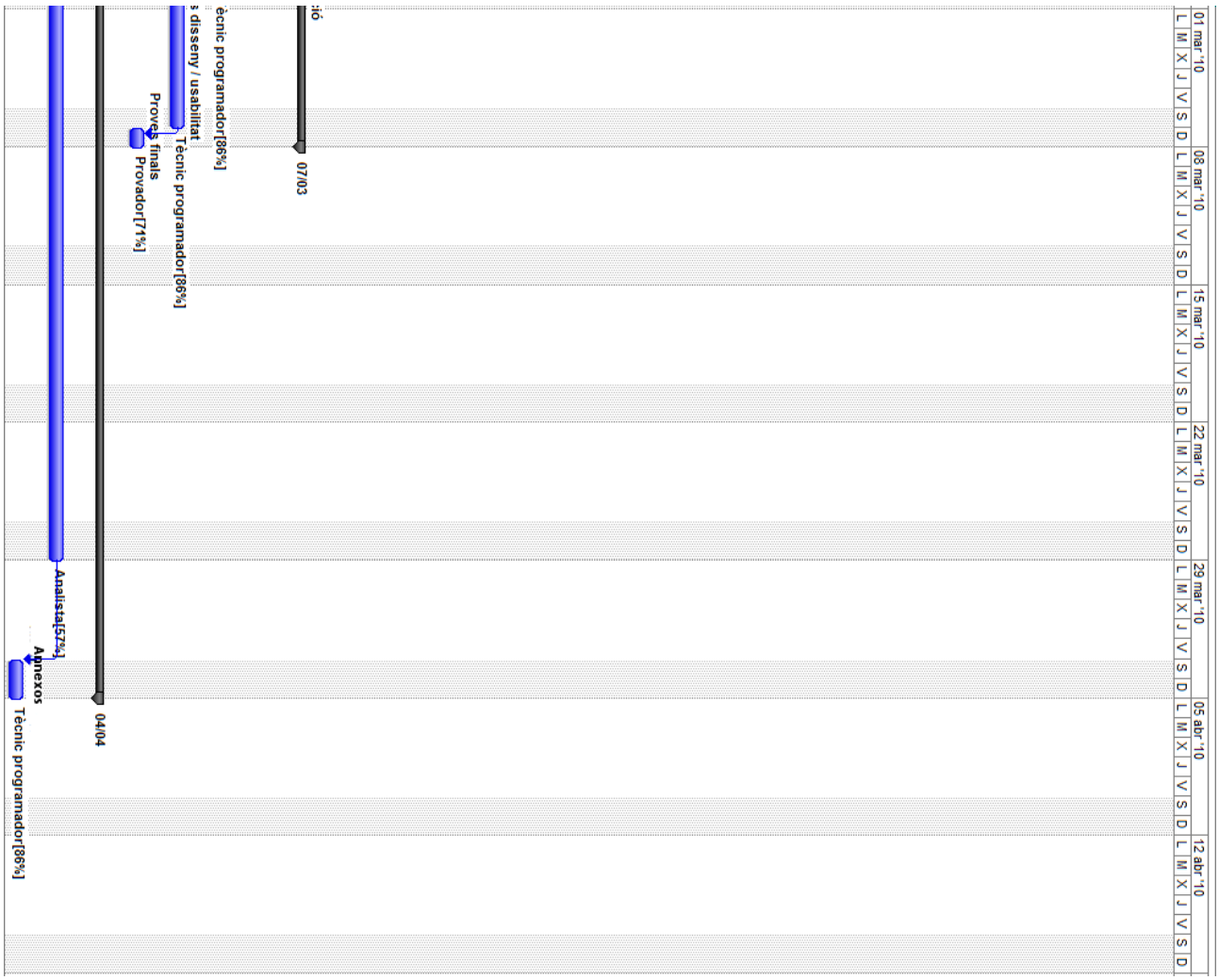
Taula 12: ús dels recursos

2.6.3 Tècniques de planificació i control

Tot seguit, es mostra la planificació realitzada mitjançant el programari *Microsoft Project 2003*. És una tasca essencial i imprescindible per programar les fases i realitzar un bon seguiment del projecte.


28 sep '09
05 oct '09
12 oct '09
19 oct '09
26 oct '09
02 nov '09
09 nov '09
16 nov '09
23 nov '09
30 nov '09
07 dic '09
14 dic '09
Anàlisi de requeriments
Descripció del projecte
Analista[43%]
Estudi d'alternatives
Analista[57%]
Anàlisis de viabilitat
Analista[86%]
Definició necessitats reals
Analista[57%]
18/10
Configuració de l'entorn de desenvolupament
Tècnic programador[29%]
Disseny
Interfície gràfica
Analista[86%]
Gestor de connexions
Analista[86%]
Interfície amb els proveïdors JMS
Analista[71%]
Funcionalitats amb destinacions i missatges
Analista[86%]
Gràfiques
Analista[86%]
29/11
Implementació
Gestor de connexions
Tècnic programador
Interfície amb proveïdors JM
Tè


21 dic '09
28 dic '09
04 ene '10
11 ene '10
18 ene '10
25 ene '10
01 feb '10
08 feb '10
15 feb '10
22 feb '10
Memòria
Documentació
Codificació
13/02
13/02
Implementació de llibreria per OpenJMS
Tècnic programador[86%]
Implementació de llibreria per Fuse
Tècnic programador[86%]
Gràfiques
Tècnic programador[86%]
Funcionalitats amb destinacions i missatges
Tècnic programador[86%]
Interfície gràfica
Tècnic programador[86%]
Proves locals
Provador[86%]
Correcció d'incidencies
Millores
Avaluació


01 mar '10
08 mar '10
15 mar '10
22 mar '10
29 mar '10
05 abr '10
12 abr '10
ió
07/03
écnic programador[86%]
s disseny / usabilitat
Tècnic programador[86%]
Proves finals
Provador[71%]
Analista[57%]
04/04
Annexos
Tècnic programador[86%]

2.7 CONCLUSIONS

El monitor de servidors JMS pretén ser una eina multiplataforma que ajudi als desenvolupadors que utilitzin la tecnologia JMS, per obtenir de forma ràpida i clara l'estat de les destinacions creades en un servidor JMS, oferint opcions com l'enviament de missatges a una destinació i la consulta del contingut dels missatges existents a una cua o tòpic, funcionalitats que s'utilitzen en el dia a dia durant les fases de desenvolupament i proves de sistemes i aplicacions que utilitzen aquesta tecnologia.

Com s'ha comentat en el primer capítol, aquest projecte té com objectius l'aprenentatge de tecnologies i la contribució d'un projecte de codi lliure per tots aquells usuaris que vulguin utilitzar una eina de monitorització de servidors JMS orientada al tràfic de missatges i programadors que vulguin millorar i ampliar aquest programa. És per tant, que els costos representats en aquest estudi de viabilitat juntament amb la factura volen mostrar el cost orientatiu del que suposaria una aplicació feta a mida amb les funcionalitats descrites.

Totes les etapes del projecte conformen el camí crític, això fa que qualsevol demora en el temps en la realització d'una tasca endarrereixi totes les restants. A més, la gran varietat de proveïdors JMS redueix la compatibilitat en aquesta primera versió del monitor, ja que per les hores valorades en el present projecte, no és possible realitzar la implementació d'un gran nombre d'interfícies que donen accés als diferents servidors fet que implicarà que els futurs usuaris que utilitzin diferents proveïdors hagin de realitzar aquesta tasca.

És cert que l'alternativa al present projecte, *Hermes JMS*, és un aplicació que permet realitzar gran part dels requeriments desitjats de forma notable i suporta una llarga llista de proveïdors JMS, però a la pràctica és una aplicació amb una interfície centrada en la monitorització del servidor i no tant en el tràfic de missatges, fet que la fa pesada a l'hora de treballar amb un gran nombre de cues i tòpics. A més els objectius i motivacions personals, la contribució amb

programari lliure i l'ús de tecnologies amb un cost nul de llicencia utilitzades en l'àmbit professional, fet que disminueix els costos del projecte, fan que el monitor de servidors JMS sigui un projecte viable.

CAPÍTOL 3 Fonaments teòrics

3.1 INTRODUCCIÓ

Amb l'arribada de Internet, la computació distribuïda ha guanyat importància en les organitzacions que cercaven crear aplicacions escalables i flexibles. Un sistema distribuït implica que diferents parts del sistema puguin estar en diferents màquines: aquestes poden ser a un mateix edifici o a un altre país.

El disseny de sistemes distribuïts no és trivial; hi ha nombrosos factors que entren en joc quan una mateixa aplicació es divideix en múltiples parts executades en diferents ordinadors; des de l'arquitectura del maquinari (*Intel, PPC*), sistemes operatius variats (*Microsoft Windows*, *Unix, Linux, OsX*) fins arribar a les pròpies comunicacions amb l'ample de banda i la velocitat de transmissió entre les màquines que conformen el sistema. Per tant, es tracta de sistemes amb una complexitat molt superior a un sistema equivalent que no ho sigui.

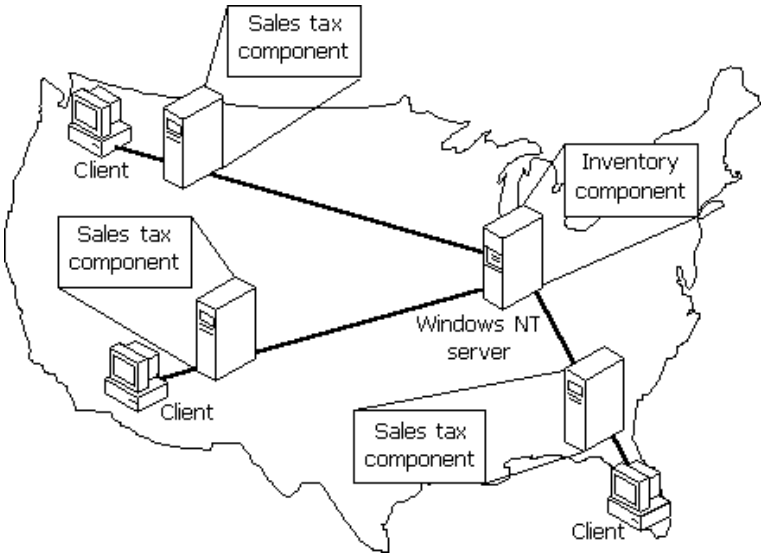


Figura 6: exemple de sistema distribuït

Un sistema distribuït pot ser dividit al menys en dos parts; una primera amb codi funcional (de negoci) i una segona amb codi dependent de la infraestructura. El model de negoci seria la funcionalitat que tracta d'assolir el desenvolupament, independentment de que el sistema sigui distribuït o no. D'altra banda, el codi d'infraestructura és totalment dependent d'aquest model d'aplicació distribuïda: si el sistema no ho és, aquesta part desapareix. Aquest és complexe i l'objectiu principal és el de transferir dades d'una part de l'aplicació a una altra. Cal dir, que aquest codi d'infraestructura depèn de la distribució i no dels objectius de negoci de l'aplicació, fet que provoca que aquest sigui una part aprofitable per altres projectes. Així doncs, quan es parla de codi d'infraestructura, es fa referència al que es coneix com *middleware*; que es podria definir com aquell component *software* que s'utilitza per connectar aplicacions entre si.

3.2 MIDDLEWARE

Middleware o programari intermediari es defineix com la capa de programari que es troba entre el sistema operatiu i les aplicacions del sistema. El principal objectiu del *middleware* és resoldre els problemes de connectivitat i interacció entre aplicacions, servint de traductor entre diferents tecnologies i protocols; és a dir, que qualsevol aplicació (independentment del seu origen) es pugui executar sota qualsevol sistema operatiu o maquinari facilitant així el desenvolupament de la mateixa i amagant detalls de programació de baix nivell. Cal dir que el programari intermediari no és imprescindible pel correcte desenvolupament d'un procés d'integració però sí que és cert que la seva utilització simplifica molt aquestes tasques.

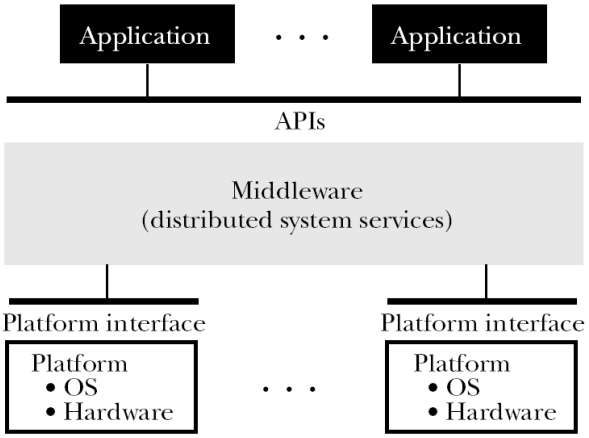


Figura 7: rol del programi intermediari

Funcions del *middleware*

- Homogeneïtzar els diferents components del maquinari, sistemes operatius i protocols de comunicació.

- En sistemes distribuïts, encapsular i ocultar les diferents parts o aplicacions connectades entre si que s'executen en diferents llocs.

- Proporcionar interfícies uniformes d'alt nivell pels desenvolupadors i integradors d'aplicacions, facilitant la composició, el reaprofitament del codi i la portabilitat d'aquestes aplicacions.

A continuació es descriuran diferents tipus de programari intermediari fent més èmfasi en l'anomenat MOM o *Message Oriented Middleware*, que és el que aplica al present projecte.

3.2.1 RPC

Remote procedure call (RPC) o *crida a procedimets remots* és una tecnologia que permet a una aplicació executar un procediment en una altra màquina present en la mateixa xarxa, sense que s'hagin de definir els detalls d'aquesta interacció remota, és a dir, el codi és el mateix tant si la subrutina és local al programa executant, o remota.

RPC neix al 1976, quan va ser descrita al RFC707. La primera implementació popular de RPC a Unix va ser la de *Sun Microsystems,* ara anomenada *Open Network Computing RPC o ONC RPC*, feta servir com a base per al sistema de fitxers en xarxa *NFS*[10]. *ONC RPC* encara es fa servir molt avui en diverses plataformes.

Una altra de les primeres implementacions de Unix va ser el *Network Computing System* (*NCS*) de *Apollo Computer. NCS* va ser utilitzat posteriorment en la fundació de *DCE/RPC* al *Entorn de Computació Distribuïda* (*DCE*) de la *Open Software Foundation (marc de treball i eines de desenvolupament per aplicacions client – servidor).* Una dècada més tard, *Microsoft* va adoptar *DCE/RPC* com a base per al seu mecanisme *Microsoft RPC (MSRPC)* i va implementar *DCOM* a sobre d'ell.

10 ***NFS***: protocol a nivell d'aplicació utilitzat com a sistema de fitxers en sistemes distribuïts en un entorn de xarxa local, possibilitant que diversos sistemes tinguin accés a fitxers remots com si fossin locals.

A mitjans dels 90, ILU de Xerox i el CORBA de *Object Management Group*, van oferir un altre paradigma de RPC basat en objectes distribuïts amb mecanisme d'herència. Cal notar que hi han moltes tecnologies *RPC* diferents que es fan servir habitualment que són incompatibles, com ara *ONC RPC* i *DCE/RPC*.

RPC permet implementar el model *client - servidor* de computació distribuïda. Un *remote procedure call* és instanciat pel *client*, enviant un missatge de petició a un *servidor* remot conegut per a executar el procediment especificat fent servir paràmetres subministrats, i mentre el servidor processa la crida, el client queda bloquejat. A continuació, una resposta és retornada al client on l'aplicació continua amb el seu procés.

Un aspecte important entre crides a procediments remots i crides locals és que les primeres poden fracassar a causa de problemes de xarxa imprevisibles. També, els clients generalment han de tractar aquests problemes sense saber si el procediment remot va ser invocat correctament.

Per a permetre l'accés de diferents clients als servidors, un nombre de sistemes RPC estanditzats ha estat creat. La majoria d'aquests fan servir un *Interface Description Language* (IDL) per a permetre a diverses plataformes cridar al RPC. Aquest és un llenguatge utilitzat per descriure la interfície de components programari: descriu una interfície en un llenguatge neutral, la qual cosa permet la comunicació entre components de programari desenvolupats en diferents llenguatges de programació. Així doncs, IDL ofereix un pont entre dos sistemes diferents. Els arxius IDL poden ser utilitzats per a generar codi que faci d'interfície entre el client i el servidor. L'eina més habitualment utilitzada per aquest propòsit és *RPCGEN*.

3.2.2 MOM i el paradigma de la missatgeria

Els ordinadors i les persones poden comunicar-se mitjançant l'ús de sistemes de missatgeria per a l'intercanvi missatges a través de xarxes electròniques. En el que es refereix a la comunicació entre aplicacions, quan es parla en termes de negoci, es refereix generalment als *Enterprise Messaging Systems* (*EMS*) o *Message Oriented Middleware* (*MOM*).

Els *EMS* permeten a dos o més aplicacions intercanviar informació a través de missatges. En aquest cas, un missatge és un paquet amb dades de negoci. Aquestes poden ser qualsevol tipus de informació i normalment conté dades sobre alguna transacció de negoci; els missatges informen a les aplicacions de que ha esdevingut un succés en altre sistema.

Amb la utilització de programari de intermediari orientat a missatges, els missatges són transmesos d'una aplicació a una altra a través de la xarxa. Els diferents productes que implementen MOM, asseguren que aquests missatges són distribuïts apropiadament entre aplicacions. A més, aquestes implementacions solen incloure funcionalitats d'alta disponibilitat, balanceig de càrrega, escalabilitat i suport transaccional per situacions on es requereixi una gran quantitat d'intercanvi de missatges.

El proveïdors de solucions MOM, utilitzen diferents formats de missatges i protocols de xarxa per l'intercanvi de dades, però els fonaments a alt nivell són els mateixos. Una API és utilitzada per crear un missatge, se li afegeix la informació en el cos, la informació de enrutament en la capçalera i s'envia. La mateixa API serà utilitzava per rebre aquest missatge produït per altres aplicacions.

En la gran majoria de EMS, l'intercanvi de missatges entre aplicacions es realitza mitjançant canals virtuals anomenats destinacions. Quan un missatge es enviat, aquest serà entregat a una destinació, no a una aplicació en concret; qualsevol aplicació que es registri o tingui una

subscripció en aquesta serà capaç de rebre el missatge. D'aquesta manera el programari que rep missatges i aquells que n'envien estan totalment desacoblats.

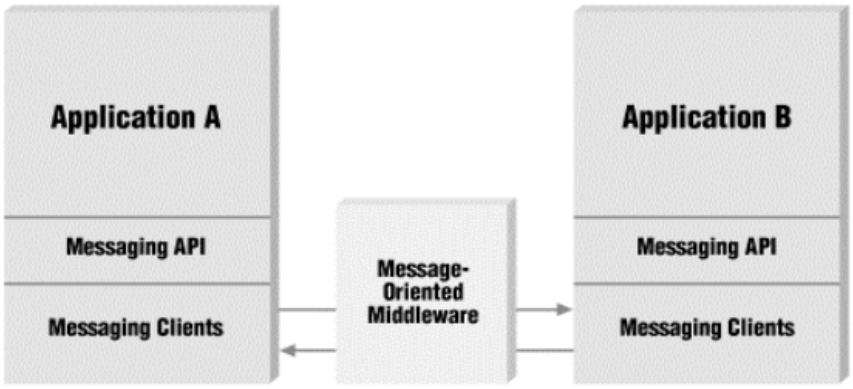


Figura 8: programari intermediari orientat a missatges

Tots el proveïdors d'implementacions basades en MOM proveeixen als desenvolupadors d'una API per l'enviament i recepció de missatges. No obstant, un dels problemes de la MOM és la falta de estàndards. La majoria de proveïdors tenen la seva pròpia implementació, API i eines d'administració i desenvolupament. Cal dir, que això no vol significa que no n'existeixin; la *Java Message Service* és un estàndard de Java que defineix una interfície per la missatgeria entre aplicacions i que la seva implementació vindrà donada pels diferents proveïdors.

Per tant, el programari intermediari MOM és una infraestructura focalitzada en enviar i rebre missatges que permet que es puguin comunicar aplicacions o mòduls diferents sobre plataformes heterogènies, reduïnt la complexitat del desenvolupament d'aplicacions que han de comunicar-se amb altres de diferents plataformes, ocultant al desenvolupador els detalls dels diferents sistemes operatius i protocols de xarxa. MOM és un programari que resideix en ambdues parts (dintre d'una arquitectura client / servidor) i que suporta trucades asíncrones. Les cues de missatges (destinacions) proveeixen de emmagatzematge temporal quan una aplicació està ocupada o no està connectada.

La missatgeria és un mètode de comunicació entre els components de programari o aplicacions. Un client pot enviar missatges a altres clients, al mateix temps que està capacitat per rebre. Cada client es connectarà a un agent o proveïdor de missatgeria que oferirà facilitats per crear, enviar, rebre, i llegir missatges. Aquest model habilita la comunicació distribuïda “imprecisa”; un component envia un missatge a una destinació i el receptor pot recuperar aquest mateix missatge quan estigui disponible per la comunicació. És per això que tant el client que envia el missatge com el que el rep, no han de estar al mateix temps preparats per la comunicació, per tant, no existeix la necessitat de que el remitent ni el destinatari es coneguin mútuament. Malgrat aquesta independència, els dos es veuen obligats conèixer el format del missatge que enviaran o rebran per tal d'utilitzar i entendre la informació continguda.

Les arquitectures MOM avui en dia varien la seva implementació; des d'una arquitectura centralitzada que depèn de un servidor de missatgeria per realitzar la distribució dels missatges, fins a una arquitectura descentralitzada que distribueix el procés de servidor entre els clients. Una gran varietat de protocols que inclouen TCP/IP, HTTP, SSL i IP *multicast* són utilitzats en la capa de transport. Alguns productes de missatgeria són híbrids dels dos models en funció de la forma en que són utilitzats.

Abans de detallar les diferents arquitectures, és important descriure el terme client; els sistemes de missatgeria estan composats per clients i un programari d'intermediari orientat a missatges. El client doncs, és una aplicació o component que utilitza la API que proporciona el proveïdor del MOM per enviar missatges.

Arquitectura centralitzada

Els EMS que utilitzen una arquitectura centralitzada confien en un servidor de missatgeria. També anomenat *broker* de missatges, és responsable de la entrega de missatges de un client als seu/s destinatari/s. La figura del servidor de missatgeria desacobla els clients que únicament

tenen visible a aquest i no als destinataris, fet que permet afegir i eliminar clients sense tenir impacte en el sistema sencer.

Típicament, una arquitectura centralitzada utilitza una topologia d'estrella. En un cas senzill, hi ha un servidor centralitzat de missatgeria i clients connectats a aquest tal com es mostra en la següent figura:

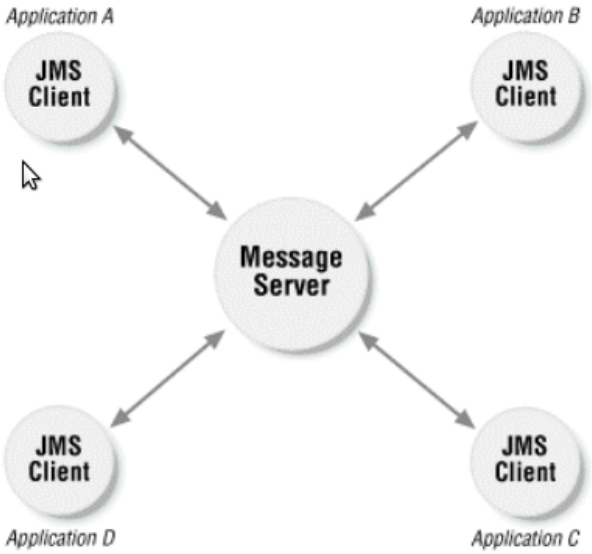


Figura 9: arquitectura centralitzada d'un MOM

Aquesta arquitectura presta per si mateixa una quantitat de connexions de xarxa mínima mentre que permet a qualsevol part del sistema comunicar-se amb qualsevol altra. En la pràctica, el servidor centralitzat acostuma a ser un clúster de servidors distribuïts operant com a una unitat lògica.

Arquitectura descentralitzada

Totes les arquitectures descentralitzades solen utilitzar IP *multicast*[11] a nivell de xarxa. Un sistema de missatgeria basat en el *multicasting* no té un servidor centralitzat. Algunes de les

11 ***Multicast***: consisteix en l'enviament de informació en un xarxa de diversos destinataris de forma simultània.

funcionalitats de un servidor (persistència, transaccions, seguretat) són incrustades com una part local del client, mentre que l'enrutament dels missatges és delegat a la capa de xarxa utilitzant el protocol IP *multicast*. Aquest permet a les aplicacions ajuntar un o més grups IP *multicast*, on cadascun utilitza una direcció IP que redistribuirà cada missatge que rebi a tots els membres del seu grup. D'aquesta manera, les aplicacions poden enviar missatges a adreces IP *multicast* confiant en que la capa de xarxa els redistribuirà correctament.

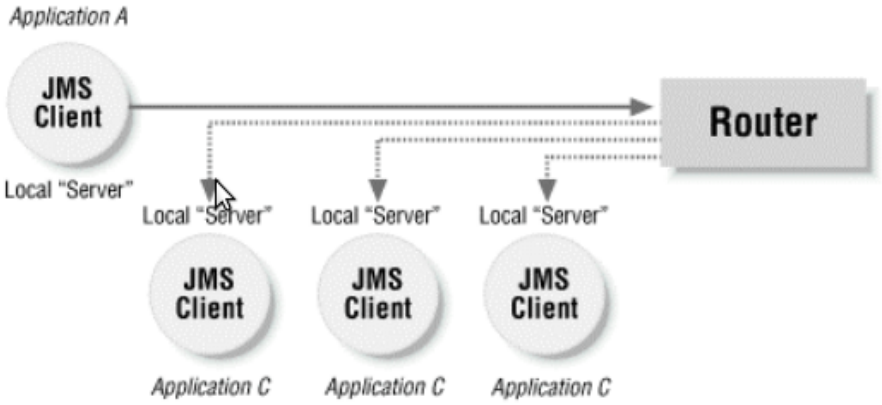


Figura 10: arquitectura descentralitzada d'un MOM

Al contrari que la arquitectura centralitzada, una arquitectura distribuïda no requereixi del servidor pel que fa a l'enrutament de missatges; la xarxa el tracta automàticament. Malgrat això, altres funcionalitats del servidor són requerides i s'inclouen en cada client, com la persistència dels missatges .

Com s'ha comentat, aquestes dues arquitectures solen utilitzar per una banda el protocol TCP/IP i per altra IP *multicast*. Entre les arquitectures implementades per diversos proveïdors també es poden trobar productes que combinen aquestes dues; on els clients poden connectar-se a un procés dimoni que utilitza TCP/IP i també a un altre que usi per la comunicació els grups IP *multicast*.

Consideracions generals sobre ambdues arquitectures.

En aquest apartat s'exposen les dos arquitectures explicades anteriorment per tal d'analitzar les avantatges i inconvenients.

Un sistema de missatgeria basat en TCP utilitza una arquitectura en estrella, on existeix la figura d'un servidor centralitzat (o un clúster amb diversos servidors) que es comuniquen amb els diferents clients utilitzant connexions TCP/IP, SSL[12] o HTTP. Aquest és responsable de qui està publicant i qui està rebent els missatges. Tenir servidors en clúster proporciona balanceig de càrrega i ajuda a optimitzar el tràfic de la xarxa filtrant i seleccionant els missatges. Els servidors s'encarregaran de la persistència de les dades, de l'accés a les destinacions i permisos dels clients per rebre els missatges. A més, els clients no necessiten saber de l'existència dels altres, només de la del servidor centralitzat.

Al mateix temps, aquesta arquitectura introdueix un únic punt de fallida; si el servidor principal de un clúster cau, el servidor sencer es tornarà indisponible. Un proveïdor de EMS pot resoldre aquest problema distribuint les connexions en servidors múltiples en un clúster, de tal forma que si un cau, l'altre servidor pot continuar operant, minimitzant l'impacte de la fallida. La reconnexió ha de ser gestionada pel client.

En canvi, el *multicasting* implica una arquitectura diferent, en la qual no existeix la figura del servidor centralitzat. És per aquest motiu que no hi ha un únic punt de fallida: cada client distribueix els missatges a la resta de clients. Una conseqüència directa d'aquest és que cada publicador i cada subscriptor han de tenir una configuració local sobre cadascun dels clients existents al sistema. Aquest fet ha de ser una consideració a tenir en compte pel desenvolupament de solucions que utilitzin el paradigma de la missatgeria; en absència de un marc de treball d'administració de alt nivell, les configuracions locals han de ser actualitzades en

12 ***SSL***: Protocol que ofereix comunicacions segures mitjançant l'encriptació de dades utilitzat amb freqüència en la web i correu electrònic .

cada client cada vegada que s'incorpora un de nou al sistema.

L'arquitectura descentralitzada necessitarà de mecanismes per garantir la persistència dels missatges, que residirà en les màquines on corren els clients. No importa com d'eficient sigui l'emmagatzematge, les escriptures i lectures de disc; aquest fet sempre serà el major coll d'ampolla del sistema. Escollint aquest tipus d'arquitectura es requereix que les màquines client tinguin un sistema d'emmagatzematge ràpid i de gran capacitat.

3.2.3 RPC vs. Missatgeria Asíncrona

Remote Procedure Call (RPC) és un terme usat per descriure un model distribuït que avui en dia és usat en tecnologies de programari d'intermediari com CORBA, Java RMI i *Microsoft* DCOM. Les tecnologies basades en RPC són una solució viable per diverses aplicacions, encara que el model de missatgeria és més optim per alguns tipus de aplicacions distribuïdes. Aquest capítol tractarà de analitzar els punts forts i dèbils de ambdós models.

RPC i la dependència modular.

Una de les àrees amb més èxit pel model RPC ha sigut la construcció d'aplicacions en 3-capes o n-capes (*3-tier, n-tier* respectivament). En aquestes aplicacions cada capa és auto-continguda de tal forma que la aplicació pot ésser dividida en diverses màquines en una xarxa distribuïda; és a dir, cada capa normalment serà mantinguda per un servidor especific per fer-se més independent de la resta. En aquest model, la capa de presentació (primera capa) es comunica utilitzant RPC amb la lògica de negoci (segona capa), la qual accedirà a la capa que emmagatzema les dades (tercera capa). La plataforma de *Sun Microsystems J2EE* i *DNA* de *Microsoft* són exemples d'aquesta arquitectura. Sense entrar en detalls en cada tecnologia i plataforma, el nucli d'aquestes és un programari basat en RPC.

Remote Procedure Call intenta imitar el comportament de un sistema que és executat en un procés; quan un procediment remot és invocat, qui truca al procediment es bloqueja fins que el procediment acaba i retorna el control a aquest. Aquest model sincronitzat permet al desenvolupador tenir una visió del sistema com si es tractes d'un sol procés. El treball es realitza de forma seqüencial, assegurant que les tasques són completades. La “sincronitzada” naturalesa de RPC augmenta la dependència dels clients (aquells components que realitzen les trucades als procediments) i el servidor (aquell qui dona servei a la trucada); el client és bloquejat fins que el servidor no respon. Aquesta dependència modular crea sistemes interdependents on una fallida en un té un impacte directe en els altres. En J2EE per exemple, el servidor EJB ha de funcionar

correctament si es vol que funcionin el *servlets*[13] que utilitzen *enterprise beans*[14].

RPC treballa correctament en diversos escenaris, però el seu sincronisme pot ser un punt dèbil on aplicacions verticals són integrades totes juntes. En aquest escenari, les línies de comunicació entre aquests sistemes verticals són diverses i multi-direccionals tal com es mostra a la següent figura:

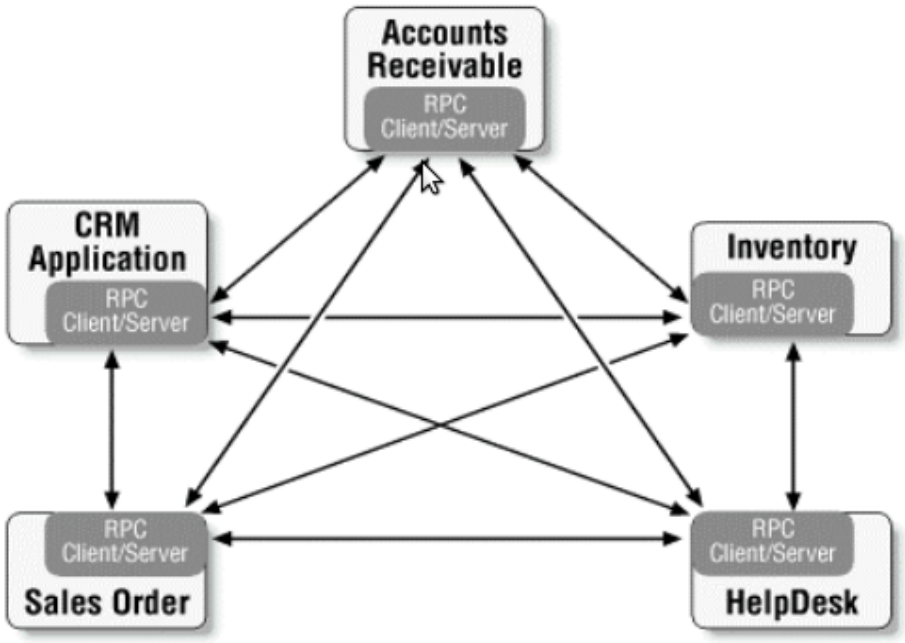


Figura 11: exemple de sistema RPC

Si es considera la implementació d'aquesta infraestructura utilitzant un mecanisme RPC existeixen diversos problemes a l'hora de gestionar les connexions entre aquests sistemes. Quan s'afegeix un altre component a aquesta infraestructura, s'haurà de tornar enrere i fer saber a la resta de sistemes la inclusió del nou. Al mateix temps, els sistemes poden fer fallida; quan una part del sistema cau, tot es paralitza. Existeixen però solucions per aquest problema com el

13 ***Servlet***: objectes que s'executen dintre d'un contenidor de *servlets*. A diferència dels *applets*, aquests corren en el servidor i no en el client.

14 ***Enterprise bean***: component de la part servidora, gestionat pel contenidor i pensat per la construcció modular d'aplicacions d'empresa.

multithreading o mecanismes CORBA, però aquestes solucions tenen força complexitat i requereixen un desenvolupament sofisticat. Els fils són costosos quan no s'utilitzen amb criteri i les trucades CORBA d'una direcció requereixen un nivell més en l'aplicació pel tractament d'errors. Per aquests motius, en escenaris on no es pot donar aquesta situació, la missatgeria pot ser considerada com alternativa.

Enterprise Messaging

Els inconvenients comentats anteriorment sobre la disponibilitat dels subsistemes no és un problema amb un programari d'intermediari orientat a missatges. Un concepte fonamental de MOM és que la comunicació entre aplicacions tendeix a ser asíncrona. El codi escrit per connectar els mòduls entre si assumeix que la resposta de l'altra aplicació no serà immediata, no és bloquejant; una vegada que un missatge és enviat, el client pot continuar realitzant altres tasques. Aquesta és la major diferència entre el RPC i la missatgeria asíncrona, on cada subsistema és desacoblat dels altres sistemes com es mostra a continuació:

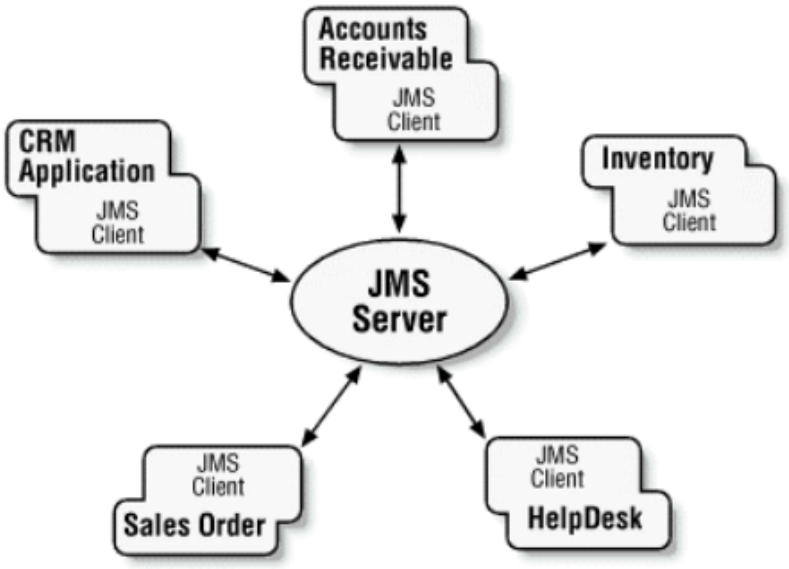


Figura 12: arquitectura Enterprise Messaging

Les diferents parts es comuniquen a través d'un servidor de missatgeria, fent que si una falla no impedeix les operacions de la resta. Una fallida no esperada o la necessitat de reiniciar un component és un fet que sol donar-se en sistemes en xarxa. La *Java Message Service*, com a exemple de MOM, garanteix l'entrega assegurant que els destinataris rebran missatges inclús quan una part del sistema cau. Aquest sistema utilitza un mecanisme d'emmagatzematge que permet que en cas de que un missatge no es pugui entregar, aquest persisteix i no es perd, tot esperant a que el seu destinatari torni a estar disponible.

En definitiva, a través del processat asíncron, la persistència i la garantia d'entrega de missatges, la *Java Message Service* proveeix funcionalitats per mantenir les aplicacions en continua operació i funcionament sense interrompre el servei. A més, ofereix flexibilitat en la integració oferint el model de missatgeria de publicador / subscriptor i el punt a punt. Mitjançant eines de control administratiu permet donar un servei robust per arquitectures basades en serveis.

3.3 JAVA MESSAGE SERVICE

3.3.1 Introducció

La *Java Message Service* (JMS) és una especificació que defineix un conjunt d'interfícies i altres semàntiques associades, les quals permeten a les aplicacions escrites en Java l'accés a serveis oferts per productes que compatibles amb JMS i el model MOM. Cal remarcar que JMS no és un producte, si no una especificació a la que productes com *MQSeries d'IBM, SonicMQ, FioranoMQ* donen la implementació.

Amb tot el potencial que els sistemes de missatgeria ofereixen, tal com s'ha comentat en apartats anteriors, han aparegut diversos productes en el mercat, cadascú amb els seus avantatges i inconvenients, més populars i menys. Com a sistemes populars de missatgeria es poden remarcar a *MQSeries* de *IBM* i *Tibco Rendezvous*. Cal dir, que aquest últim no dona suport a JMS, però si ho fa un altre producte de la mateixa companyia anomenat *Tibco EMS*. Tots aquests productes tenen les seves pròpies interfícies i API i són entre ells lleugerament diferents.

Es pot considerar la següent situació; un client que desenvolupa una aplicació requereix un producte de missatgeria i crea una llista de requeriments. Després d'avaluar les diferents alternatives existents, aquest client selecciona el proveïdor que millor s'ajusta als seus requeriments i el client acaba per integrar aquest producte al seu desenvolupament. Un any després, altre proveïdor ofereix un producte millor o apareixen nous requeriments que el producte actual no assoleix. El client no pot migrar fàcilment el seu desenvolupament a un nou producte degut a la forta dependència entre el producte de missatgeria i el seu codi. És en aquest punt on entra en escena la *Java Message Service*, oferint un conjunt uniforme de interfícies i semàntiques per sistemes de missatgeria. Per tant, permet als clients tenir una visió uniforme de tots els productes que acompleixen amb aquest estàndard facilitant el canvi de proveïdor de missatgeria i minimitzant els costos d'una migració.

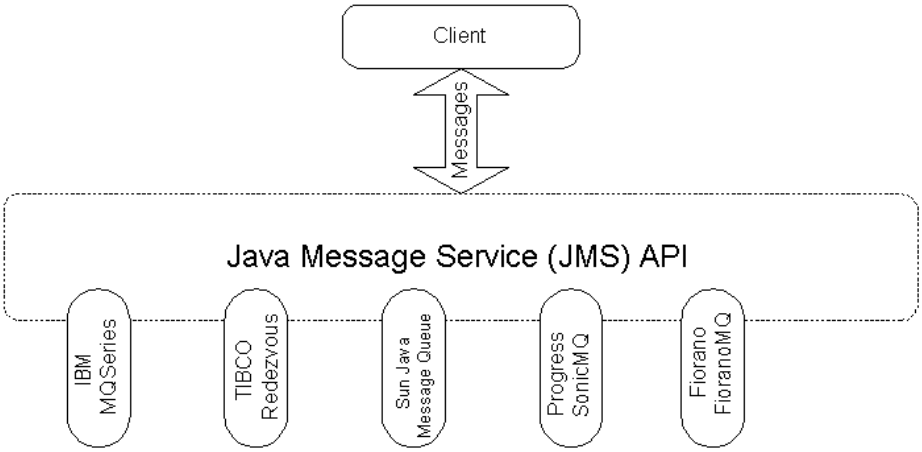


Figura 13: interfície JMS i proveïdors

Com es mostra en la figura anterior, aquest model allibera al client de l'ús d'un únic proveïdor i amplia les diferents propostes en el mercat. JMS té com objectiu proporcionar una API uniforme de missatgeria.

Un nombre important d'empreses com *Allaire, BEA Systems, Fiorano Software, Progress Software* van col·laborar en un inici amb *Sun Microsystems* per definir un primer esborrador de la especificació JMS. A més, es van rebre idees i comentaris d'altres companyies i organitzacions educatives durant tres mesos de revisió pública. Aquesta és una de les claus de l'acceptació de JMS, a part òbviament de les avantatges comentades anteriorment. Actualment empreses com *Oracle, Sybase, Novel* i *IBM* patrocinen l'especificació JMS.

És important entendre que JMS no representa la unió de funcionalitats disponibles a través dels diferents productes existents; la especificació seria massa voluminosa per a que qualsevol proveïdor donés suport. Al mateix temps seria complicada pels desenvolupadors obtenir una comprensió ràpida, ja que la corba d'aprenentatge creixeria. D'altra banda, JMS no és una intersecció de un conjunt de funcionalitats de productes de missatgeria que existien amb anterioritat; en comptes es va definir un conjunt de conceptes de missatgeria i facilitats que són

bàsiques per implementar una aplicació de missatgeria. Es pot prendre de forma anàloga a JDBC[15], on aquest permet l'accés uniforme a diferents tipus de bases de dades relacionals mentre que JMS ho fa amb els diferents productes de missatgeria.

3.3.2 Models (publicador - subscriptor / punt a punt)

El model de publicador / subscriptor permet al productor del missatge (també anomenat publicador) fer arribar el seu missatge a més d'un consumidor o destinatari (anomenats subscriptors). Cal tenir en compte tres aspectes importants d'aquest model:

- Els missatges són extrets del servidor cap als consumidors sense que ells tinguin una petició sobre de rebuda d'aquests. Els missatges són intercanviats mitjançant un canal virtual anomenat tòpic. Un tòpic és una destinació on els productors poden publicar i els subscriptors poden consumir missatges. Els missatges entregats en tòpics són automàticament extrets a tots els consumidors subscrits.

- No hi ha dependència entre productors i consumidors. Subscriptors i publicadors poden ser afegits dinàmicament en temps d'execució, el que permet al sistema créixer o fer-se més petit durant el cicle d'execució.

- Cada client que subscrit a un tòpic rep la seva copia del missatge publicat en un tòpic. Un missatge produït per un publicador pot ser copiat i distribuït a centenars o milers de subscriptors.

D'altra banda, en el model punt a punt, el productor és anomenat remitent i el consumidor destinatari. Les característiques més importants d'aquest model són:

15 ***JDBC***: *Java DataBase Connectivity* permet a les aplicacions en llenguatge Java accedir mitjançant una interfície comuna a les bases de dades per a les que existixen *drivers* JDBC. Normalment, es tracta de bases de dades relacionals.

- El missatges són intercanviats a través de un canal virtual anomenat cua. Una cua és una destinació que permet als remitents enviar missatges i als destinataris obtenir-los.

- Cada missatge és només entregat a un destinatari. Es poden connectar diversos consumidors a una cua, però cada missatge només serà entregat a un d'ells.

- Els missatges són ordenats. Una cua entrega els missatges a un client consumidor en l'ordre en que foren enviats pel productor. Quan un missatge es consumeix, s'elimina del cap de la cua.

- No hi ha dependència entre productors i consumidors. Destinataris i remitents poden ser afegits dinàmicament en temps d'execució (al igual que els tòpics).

En la majoria de casos, la decisió de quan utilitzar un model o un altre dependrà dels avantatges que ofereixen ambdós. Amb subscriptors i publicadors, es pot tenir qualsevol nombre de subscriptors en un tòpic i tots rebran una copia del mateix missatge. El publicador no tindrà en compte si hi ha algú escoltant o no, per exemple; un client publicador envia informació sobre quotes. Si cap subscriptor en particular no està connectat no rebrà mai aquest missatge i el publicador no se'n donarà compte. D'altra banda, una sessió punt a punt està enfocada en una conversació entre dos components. En aquest escenari cada missatge és realment important. El rang i la varietat de dades que els missatges representen pot ser un factor a tenir en compte també. Utilitzant el model de publicador i subscriptor, inclòs quan la missatgeria està sent utilitzada per establir una conversa entre dos components que es coneixen, pot ser avantatjós utilitzant diversos tòpics per diferenciar el tipus de missatges que s'estan enviant. Cada tipus de missatge pot ser interpretat per separat a través d'un únic consumidor. No obstant, el punt a punt és més convenient quan es vol assegurar que un destinatari processarà aquest missatge una sola vegada.

Aquestes són les diferencies crítiques dels dos, el punt a punt assegura que només un consumidor tractarà aquell missatge. Aquest fet és important quan els missatges han de ser processats separadament però en tàndem, balancejant la carrega a través de diferents clients JMS. Una altra avantatge és que el model punt a punt proveeix un *QueueBrowser*[16], que permet al client accedir a la cua per llegir els missatges que estan esperant ser consumits sense consumir-los.

Cal afegir, que existeix un sistema a JMS per garantir la persistència dels missatges que s'envien a través de tòpics; les subscripcions durables. Aquestes guarden els missatges quan el subscriptor és inaccessible per evitar la pèrdua del missatge en clients que utilitzin aquest model i que no es puguin permetre perdre missatges.

3.3.3 Fonaments bàsics de JMS

En aquest apartat es fa una descripció dels conceptes bàsics de JMS per tal d'entendre certs aspectes del projecte que s'exposa. S'adjuntarà codi Java per mostrar un exemple d'implementació de client JMS el qual utilitzarà les següents variables:

```
String                  queueName = null;
Context                 jndiContext = null;
QueueConnectionFactory  queueConnectionFactory = null;
QueueConnection         queueConnection = null;
QueueSession            queueSession = null;
Queue                   queue = null;
QueueSender             queueSender = null;
TextMessage             message = null;
final int               NUM_MSGS;
```

16 ***QueueBrowser***: Interfície de la API de JMS. Les classes que la implementen proporcionen visibilitat als elements d'una cua però sense treure'ls de la mateixa, i per tant, continuen en el servidor sense ser consumits.

Connection Factories

En el centre del nucli de JMS, una connexió representa un enllaç lògic amb els proveïdor de JMS. És obvi que una de les primeres coses que ha de realitzar un client JMS és establir una connexió amb un servidor JMS. Per obtenir aquesta connexió cada proveïdor JMS facilita una *connection factory*. JMS no estandarditza la informació que conté aquesta *connection factory* o com el client obté la aquesta connexió de un proveïdor de JMS.

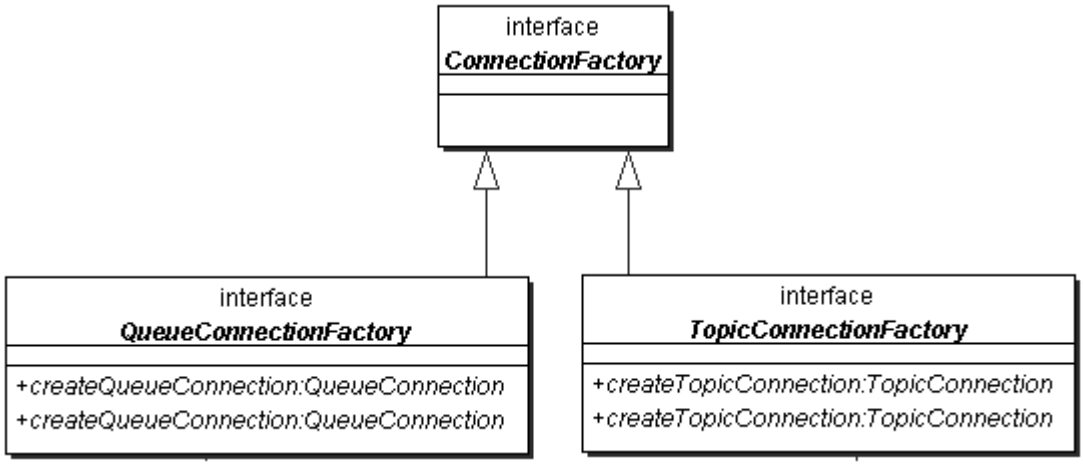


Figura 14: interfície ConnectionFactory

Tal com es mostra a la figura anterior, existeixen dos tipus de *connection factories*: una per la connexió punt a punt i una altra pel model publicació / subscripció. Basat en el model de missatgeria que utilitzaran, els clients obtindran la *connection factory* apropiada i es connectaran al proveïdor JMS.

```
/*
 * Create a JNDI API InitialContext object if none exists
  * yet.
 */
      try {
          jndiContext = new InitialContext();
      } catch (NamingException e) {
          System.out.println("Could not create JNDI API " +
              "context: " + e.toString());
```

```
            System.exit(1);
        }

        /*
         * Look up connection factory and queue.  If either does
         * not exist, exit.
         */
        try {
            queueConnectionFactory = (QueueConnectionFactory)
                jndiContext.lookup("QueueConnectionFactory");
            queue = (Queue) jndiContext.lookup(queueName);
        } catch (NamingException e) {
            System.out.println("JNDI API lookup failed: " +
                e.toString());
            System.exit(1);
        }
```

En el fragment de codi anterior, es pot observar com es busca la *factory* mitjançant JNDI. Aquesta és la forma correcta d'establir una connexió amb un servidor JMS. Normalment els proveïdors tenen mètodes propis dependents de la seva implementació i que no figuren a la definició JMS. És per aquest motiu que si es vol fer un codi totalment independent del proveïdor s'ha d'utilitzar aquest sistema.

Sessions

Una vegada que el client ha establert una connexió amb el proveïdor JMS, el següent pas es començar una nova sessió; una vista privada del client sobre la connexió. Cada connexió pot tenir diverses sessions obertes al mateix temps. Al mateix temps que una connexió és necessària per comunicar-se amb el proveïdor JMS, una sessió serà necessària per comunicar-se amb la mateixa connexió. Per entendre millor aquest concepte es pot posar un exemple: la connexió seria anàloga a la línia de telefon que fa de servei a tot el barri, mentre que la sessió seria la trucada telefònica que utilitza aquesta línia.

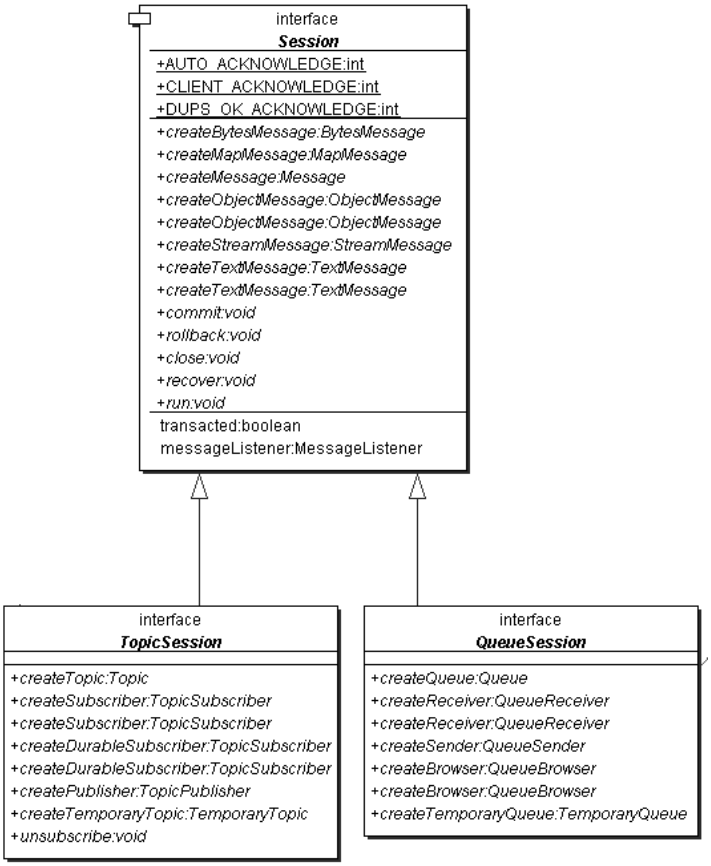


Figura 15: interfície Session

Al establir una sessió es defineix el tipus de *acknowledge* que s'utilitzarà per especificar quina serà la forma de confirmar que la transacció d'un missatge ha sigut correcta. A JMS es contemplen tres maneres:

- **mode automàtic:** quan una sessió empra aquest mode, els missatges enviats o rebuts d'aquesta sessió automàticament són reconeguts (*acknowledged*).

- ***duplicates okay mode***: els missatges rebuts o enviats d'una sessió són automàticament reconeguts com en el mode anterior, però no al moment exacte. Aquest fet provoca que a vegades els missatges puguin ser entregats més d'una vegada.

- **mode client:** en aquest cas l'aplicació serà l'encarregada de confirmar que el missatge ha sigut enviat o rebut correctament, donant-li control complet però augmentant la complexitat

del codi.

Destinacions

En un sistema basat en missatgeria, sense tenir en compte el model que s'estigui utilitzant (publicar/subscriure o punt a punt), cada missatge ha de ser entregat en algun lloc, que en JMS es coneix com destinació. Els missatges són enviats una destinació, i són rebuts d'una destinació al mateix temps. JMS no estandarditza quina és la informació que encapsula la destinació.

Hi ha dos tipus de destinacions en funció del model de enviament i recepció que s'estigui utilitzant. Pel model punt a punt, la destinació s'anomena cua i per la publicació i subscripció tòpic. Una sessió creada per un model punt a punt únicament pot ésser utilitzada per arribar a una cua, fet que aplica també a les sessions creades per treballar amb tòpics.

Una vegada el client JMS ha obtingut una sessió mitjançant el establiment d'una connexió, a través de la sessió s'obté la destinació. És en aquest moment on el client ja podrà enviar missatges a aquell destí. Malgrat això, la sessió mateixa no pot ser utilitzada per rebre i enviar missatges si no que actua com una *factory* que pot ser utilitzada per crear remitents i destinataris que són utilitzats per enviar i rebre missatges respectivament.

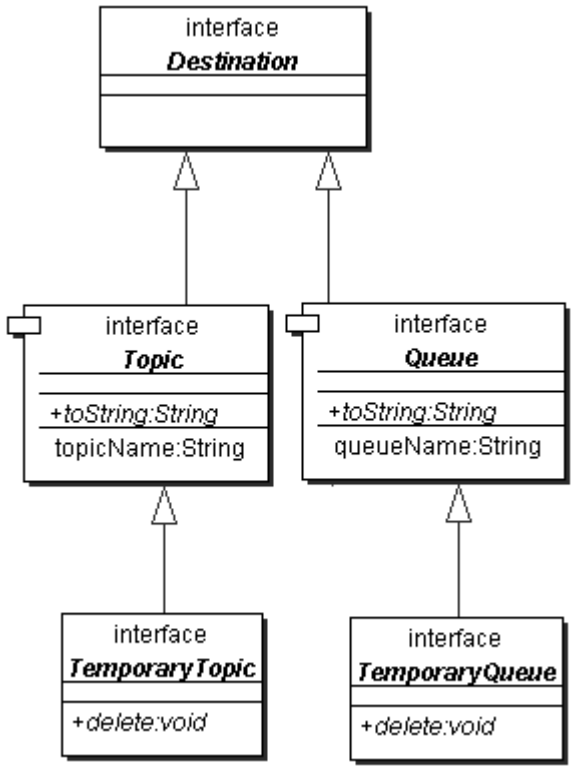


Figura 16: interfície Destination

Una vegada finalitzada la connexió, s'ha de tancar. En el següent fragment de codi, es mostra aquest procés:

```
/*
 * Create connection.
 * Create session from connection; false means session is
 * not transacted.
 * Create sender and text message.
    * Send messages, varying text slightly.
    * Send end-of-messages message.
    * Finally, close connection.
    */
       try {
           queueConnection =
               queueConnectionFactory.createQueueConnection();
           queueSession =
               queueConnection.createQueueSession(false,
                   Session.AUTO_ACKNOWLEDGE);
           queueSender = queueSession.createSender(queue);
           message = queueSession.createTextMessage();
```

```
            for (int i = 0; i < NUM_MSGS; i++) {
                message.setText("This is message " + (i + 1));
                System.out.println("Sending message: " +
                    message.getText());
                queueSender.send(message);
            }
        } catch (JMSException e) {
            System.out.println("Exception occurred: " +
                e.toString());
        } finally {
            if (queueConnection != null) {
                try {
                    queueConnection.close();
                } catch (JMSException e) {}
            }
        }
```

El missatge

El missatge és la part més important de la especificació JMS. Totes les dades i esdeveniments en una aplicació JMS són comunicats mitjançant aquest medi, mentre que la resta dels components JMS existeixen per facilitar la transferència d'aquest. En sistemes basats en RPC (*CORBA, Java RMI, DCOM*), un missatge és una comanda per executar un mètode o un procediment, el qual bloqueja al remitent fins que no rep una resposta. Un missatge JMS no és una instrucció; transfereix les dades i diu al destinatari quelcom que ha passat. Aquest fet aïlla al destinatari del remitent fent el sistema més dinàmic i flexible que aquests anteriors. Un missatge té dos parts; el cos del missatge amb les dades, la capçalera i les propietats.

La capçalera proveeix informació sobre el missatge descrivint qui i què a creat el missatge, quan ha sigut creat, la vigència de les dades, etc. La capçalera també inclou informació d'enrutament que descriu el destí del missatge (tòpic o cua), com el missatge serà reconegut (*acknowledge*) entre d'altres. A més poden també incloure propietats definides pel client JMS. La majoria de vegades les capçaleres són automàticament inicialitzades, els seus valors són assignats pel proveïdor JMS quan el missatge es entregat, per tant alguns dels valors definits pel

desenvolupador mitjançant el mètode *setJMS<HEADER>()* són ignorats.

Capçaleres d'assignació automàtica	Capçaleres d'assignació manual
JMSDestination	JMSReplyTo
JMSDeliveryMode	JMSCorrelationID
JMSMessageID	JMSType
JMSTimestamp	
JMSExpiration	
JMSRedelivered	
JMSPriority	

Taula 13: capçaleres dels missatges JMS

Les propietats actuen com capçaleres addicionals que poden ser assignades a un missatge. Proveeixen als desenvolupadors més informació sobre el propi missatge. La interfície *Message* dona mètodes d'accés i escriptura per establir propietats. Es donen tres categories bàsiques de propietats en els missatges; especifiques de l'aplicació, definides per JMS i definides pel proveïdor. Les primeres són definides i aplicades als missatges pel desenvolupador del client, la segona i la tercera són opcionals i són en major part afegides pel proveïdor JMS.

JMS contempla sis tipus de missatges que han de suportar els proveïdors, on especifica les seves interfícies però no la implementació; això fa que els proveïdors tinguin llibertat per implementar el transport dels missatges tot mantenint-se transparent al codi del desenvolupador JMS. En total són sis interfícies destinades al missatge on cinc són subinterfícies; *TextMessage, StreamMessage, MapMessage, ObjectMessage* i *BytesMessage*. Aquestes estan definides en funció del missatge que transportaran.

TIPUS	DESCRIPCIÓ
TextMessage	El cos és una cadena de text. Són útils per intercanviar dades en forma de caràcters o enviar XML. *...* *TextMessage textMessage = session.createTextMessage( );* *textMessage.setText("Hello!");* *topicPublisher.publish(textMessage);* *...* *TextMessage textMessage = session.createTextMessage("Hello!");* *queueSender.send(textMessage);* ...
StreamMessage	Aquest missatge transporta un *stream* de tipus primitius de Java (*int, double, char...*). Proveeix un conjunt de mètodes pel mapeig de *stream* de *bytes* en dades primitives de Java. ... *StreamMessage streamMessage = session.createStreamMessage(* *streamMessage.writeLong(2938302);* *short value = streamMessage.readShort( );* *...*
MapMessage	El cos transporta un conjunt de parelles (nom-valor). La clase MapMessage és útil per entregar dades referenciades per claus. ... *MapMessage mapMessage = session.createMapMessage( );* *mapMessage.setInt("Age", 88);* *mapMessage.setFloat("Weight", 234);* *mapMessage.setString("Name", "Smith");* *mapMessage.setObject("Height", new Double(150.32));* *....*
ObjectMessage	El missatge conté un objecte *Serializable*[17] de Java. És útil per l'intercanvi d'objectes. *...* *Order order = new Order( );* *...* *ObjectMessage objectMessage = session.createObjectMessage( );* *objectMessage.setObject(order);* *queueSender.send(objectMessage);* *...*

17 ***Serializable***: API de Java que permet la gestió de la serialització d'objectes. S'entén per serialització d'objectes la conversió en *bytes* d'aquests per ser a posteriori llegits i restaurats.

	ObjectMessage objectMessage = session.createObjectMessage(order); *topicPublisher.publish(objectMessage);* *...*
BytesMessage	El cos del missatge conté un vector format pel tipus primitiu byte. Útil per intercanviar informació entre aplicacions que no suporten els altres tipus de missatges. ... *BytesMessage bytesMessage = session.createBytesMessage( );* *bytesMessage.writeChar('R');* *bytesMessage.writeInt(10);* *bytesMessage.writeUTF("this is an example");* *queueSender.send(bytesMessage);* *...*

Taula 14: tipus de missatges JMS

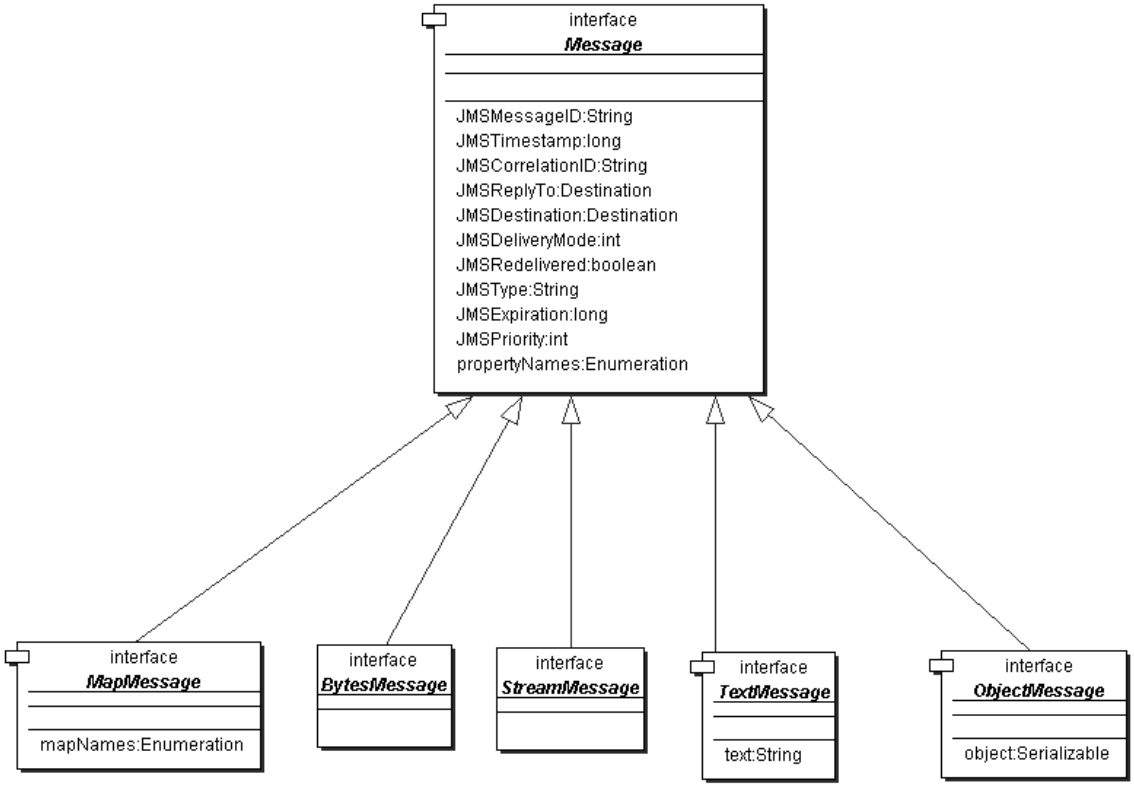


Figura 17: interfície Message

Transaccions

Les transaccions en JMS compleixen un conjunt de propietats conegudes com *ACID*, acrònim que significa *atomicity, consistency, isolation, durability*. Quan es parla d'atomicitat es refereix a que s'envien o reben tots els missatges d'una transacció o cap. La consistència significa que tots els missatges d'una transacció són consistents. Aïllament vol dir que encara que existeixen diverses transaccions en un sistema, aquestes no s'afecten entre sí. Per últim, la durabilitat es defineix com que quan una transacció és confirmada, tots els canvis es fan efectius i sobreviuen a fallides del sistema.

El suport de les transaccions de JMS es construeix en l'objecte sessió, el qual es podrà especificar com transaccional :

```
QueueConnection connection = // obtenir la connexió
QueueSession session = null;

session = connection.createQueueSession(true, Session.AUTO_ACKNOWLEDGE);
```

En aquest cas, el primer paràmetre de la trucada a *createQueueSession* és “true”, indicant així que una sessió transaccional es requerida. La sessió transaccional agrupa els missatges produïts en una unitat atòmica de treball. Quan la transacció és confirmada (*commit*) els missatges són confirmats (*acknowledged*) com una única unitat i els missatges s'envien. Si la transacció és cancel·la (*rollback*), els missatges produïts són destruïts i els missatges consumits són automàticament recuperats.

JMS no requereix que un proveïdor implementi transaccions per ser acomplir amb XA[18]. Les transaccions que acompleixen XA segueixen dues fases a l'hora de ser confirmades. Com a exemple d'aquest fet, *Fiorano MQ 3.0* que acompleix amb l'estàndard JMS, no acompleix amb XA.

18 ***XA***: Especificació pel processament de transaccions distribuïdes. Descriu la interfície entre el gestor de transaccions global i el gestor de recursos local. Els gestors de recursos que segueixen aquesta especificació s'anomenen com *XA compliant.*

De fet JMS tampoc requereix que els seus proveïdors suportin transaccions distribuïdes. No obstant si un hi pot treballar, ho haurà de fer mitjançant la API *JTA XA Resource*[19]. La gran majoria de productes que implementes JMS no suporten aquest tipus de transaccions.

Duplicació de missatges

La especificació JMS obliga a que un proveïdor JMS mai entregui una segona copia de un missatge ja reconegut. Quan un client utilitza una sessió en mode automàtic (AUTO_ACKNOWLEDGE), no té el control directe d'aquesta confirmació. D'aquesta manera aquests no poden saber amb certesa si el missatge ha sigut rebut i per tant continuen preparats en cas de que s'hagi de tornar a entregar aquest.

Un altre norma és que els proveïdors no han de produir missatges duplicats. Això significa que el productor ha de confiar en que el proveïdor JMS farà arribar aquest missatge als consumidors.

Multithreading

El *multithreading* està inclòs dintre de la plataforma Java profundament, és a dir, ofereix una forma simple, elegant i potent per crear programes que utilitzin fils. JMS classifica els objectes en dos categories, aquells que són compartits per clients *multithread* i aquells que són accedits utilitzant un únic fil de control cada vegada. Els objectes que suporten concurrència són *Destination, ConnectionFactory* i *Connection*. *Session, MessageProducer* i *MessageConsumer* en canvi no.

Existeixen dues raons per restringir l'accés concurrent a les sessions. Un és el suport a les transaccions, que són difícils d'implementar quan es treballa amb múltiples fils. L'altre és que les

19 ***JTA:*** La *Java Transaction API* és una de les API de *JEE* que permet transaccions distribuïdes a través de diversos recursos XA en un entorn Java.

sessions suporten el consum asíncron de missatges. Si les sessions suportessin l'accés concurrent, els clients haurien de codificar els seus propis gestors de missatges asíncrons per tal de ser capaços de gestionar múltiples missatges concurrents.

3.3.4 Escenaris

En aquesta secció s'intenta descriure alguns escenaris del món real per donar una idea de quin són els tipus de problemes que la tecnologia JMS pot esdevenir una eina important.

Integració d'aplicacions

La major part d'organitzacions tenen aplicacions que utilitzen des de fa anys i altres que són noves. Pot existir doncs una necessitat d'integrar aquestes antigues aplicacions amb les noves per tal de compartir dades i cooperar a fi de poder millora el servei i l'efectivitat. La integració d'aquest tipus d'aplicacions generalment s'anomena *Enterprise Application Integration* (EAI).

Existeixen diverses solucions per aquest propòsit però sense dubte els sistemes de missatgeria són una bona solució ja que permeten l'intercanvi de dades mantenint la independència dels diferents sistemes. Amb tòpics i cues es poden informar d'esdeveniments o simplement enviar dades que poden ser tractades per l'altra part. Com exemple, un sistema de missatgeria podria ser utilitzat per integrar una comanda realitzada per Internet amb un *Enterprise Resource Planning* (ERP[20]) com SAP, on la aplicació que gestiona la comanda enviaria les dades de negoci al ERP mitjançant un tòpic o una cua.

20 ***ERP***: Sistema de gestió de la informació que integra i automatitza moltes de les pràctiques de negoci associades als aspectes operatius i productius d'una empresa.

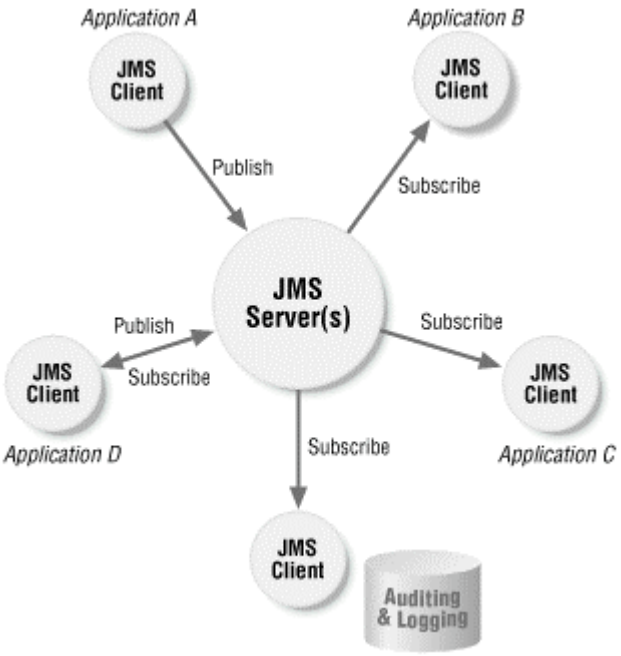


Figura 18: JMS en la integració d'aplicacions

Dispersió geogràfica

Avui en dia moltes companyies es troben dispersades geogràficament; sistemes destinats a la gestió d'inventaris han de comunicar-se amb aplicacions ERP centralitzades, dades sensibles d'empleats que són administrades localment en cada subdelegació necessiten estar sincronitzades amb les de la delegació principal... JMS pot ajudar a assegurar l'intercanvi de dades a través d'un model geogràfic distribuït d'una corporació.

3.3.5 Consideracions d'una implantació amb JMS

Rendiment i escalabilitat

Existeixen una gran varietat de productes que implementen l'estàndard JMS i no és una tasca senzilla escollir un que s'adapti a les necessitats reals del sistema que es vol implementar. El rendiment i la escabilitat són termes que es solen ajuntar, però cal distingir-los. El rendiment fa referencia a la velocitat en que un proveïdor JMS pot processar missatges, en canvi la escalabilitat al nombre de clients connectats de forma concurrent que aquell servidor JMS pot suportar. Així

doncs, una prova amb dos o tres clients enviant i rebent missatges amb una altra amb milers de clients poden determinar quin és el rendiment i la escabilitat d'aquella implementació de JMS.

Es important provar directament el producte tot realitzant un conjunt de test per veure quin és el que s'adapta millor als requeriments del sistema. Abans però, s'hauran de tenir en compte certs aspectes:

- La mida de la aplicació o solució a desenvolupar. És important fer una predicció del creixement que tindrà en el temps.
- La carrega mitjana que requerirà la aplicació.
- Els màxims de càrrega que es poden produir al desenvolupament. Pot haver-hi certs dies o hores al dia on l'intercanvi de missatges sigui més intents.
- El nombre de connexions que utilitzarà l'aplicació.
- La quantitat de dades que han de processar-se a través del sistema de missatgeria en un període de temps. Es pot mesurar per segons, bytes per segon, missatges per mes...
- La mida que tindran els missatges.
- Ús de cues o tòpics; si l'aplicació utilitzarà cues segurament el rendiment es veurà afectat ja que aquestes asseguren la persistència i per tant els missatges normalment són escrits a disc, fet que fa baixar els ràtios de missatges entregats per segon si es compara amb un tòpic.

Com ja s'ha comentat, provar amb un petit nombre de clients no és el mateix que amb cents o milers. Una bona pràctica és veure el comportament del sistema en funció d'increments. Per exemple es pot iniciar un test de 10 productors i 10 consumidors i 100.000 missatges. Després es pot incrementar a 100 productors i 100 consumidors i 1.000.000 de missatges.

També és important saber les necessitats reals de maquinari; si la CPU o la memòria utilitzada no arriba als màxims i el ràtio de missatges no millora, segurament és perquè l'escriptura a disc (sobretot en missatgeria persistent) o la xarxa és el coll de ampolla del sistema. Per tant serà important disposar d'un sistema de fitxers adequat i discs ràpids en operacions d'escriptura / lectura, o d'un ample de banda superior en el cas del segon.

Multicast

En el protocol TCP/IP (*Transmission Control Protocol / Internet Protolcol*), un procés que vulgui establir una comunicació amb un o més processos a través de la xarxa, crea una connexió per cadascun d'aquest processos, enviant i rebent dades usant aquestes connexions. Aquest protocol assegura que totes les dades arriben en el ordre correcte a més de descartar les dades duplicades. Si esdevé un succés no esperat amb la connexió, immediatament els processos que es comuniquen coneixen immediatament l'estat de la connexió.

En canvi en el protocol UDP (*User Datagram Protoco*l) , les dades són enviades a un destí però no s'assegura que aquestes arribaran. El procés que rep les dades mai sap que el remitent envia dades. Així doncs, la responsabilitat de que els processos que envien dades entre ells mitjançant UDP s'assabentin de que la informació ha sigut enviada correctament recau en les mateixes aplicacions. No obstant, aquest comportament de UDP fa realitat un tipus de servei que és completament diferent; TCP està fonamental limitat per comunicacions punt a punt. UDP ofereix la noció *multicast*, en la que una aplicació pot enviar dades a un grup de destinataris. El *multicasting* es basa en una classe especial d'adreces conegudes com classes D. Aquestes no són assignades a cap destí en concret, si no a grups *multicast*. Les dades enviades a una adreça *multicast* només seran rebudes per aquell grup *multicast*, per tant, l'opció *multicast* des d'un punt de vista de la xarxa, sembla l'opció més eficient quan es parla d'enviar un missatge a diversos destinataris.

Un sistema de missatgeria basat en TCP, normalment utilitzarà una arquitectura d'estrella on existirà al centre un servidor de missatgeria o un clúster de servidors, que es comunicaran amb els clients utilitzant TCP/IP, SSL o connexions HTTP. El servidor serà l'encarregat de saber quin client envia i quin rep en cada moment. A més s'introdueix un únic punt de fallida, si el servidor principal cau, tot el sistema cau.

No obstant, si un sistema de missatgeria es basa en *multicasting*, implica una arquitectura diferent on desapareix la figura del servidor centralitzat de missatgeria. Com no hi ha servidor central, no hi ha un únic punt de fallida; cada client envia directament als altres clients. Una conseqüència d'aquest model és que cada productor i cada consumidor hauran de tenir una configuració local sobre els dos en el sistema, fet que pot ésser complicat d'administrar quan tenim desenes, centenes o milers de clients. A més, la persistència dels missatges recaurà en els clients i dependrà dels seus sistema de fitxers i discs.

IP *multicast* aporta millores significatives en la càrrega de dades sobre la xarxa en missatges d'un a molts. Un missatge *multicast* a múltiples destinacions implicarà menys trafic en la xarxa que enviar un missatge per cada client utilitzant connexions TCP. Malgrat aquest fet, l'opció de quin sistema escollir no és trivial; el rendiment de IP *multicast* únicament és viable per un tipus de desenvolupaments depenent del tipus de missatges que s'utilitza, el maquinari de xarxa, l'entorn del desenvolupament (i*ntranet, internet*) i la complexitat de l'administració, juntament amb altres aspectes que s'han vist a aquest apartat.

Seguretat

Els proveïdors de JMS implementen les seves solucions donant mecanismes per garantir la seguretat i l'accés als missatges. L'autenticació verifica la identitat del usuari que intenta accedir al sistema de missatgeria; el client JMS s'haurà d'identificar per establir una comunicació amb el servidor. L'autenticació està suportada en la API de JMS quan es crea un objecte *Connection*, a

més de la API de JNDI com es mostra a continuació:

```
Properties env = new Properties( );
env.put(Context.SECURITY_PRINCIPAL, "username");
env.put(Context.SECURITY_CREDENTIALS, "password");
TopicFactory topicFactory = jndiContext.lookup("...");
...
TopicConnection con =
  topicFactory.createTopicConnection("username", "password");
```

A més, els proveïdors JMS també poden utilitzar sistemes més complexes per l'autenticació com sistemes de clau privada i pública. Però l'autenticació només és el primer pas, una vegada s'ha accedit a la connexió és poden establir permisos sobre les accions que es poden realitzar en aquest servidor; l'autorització aplica polítiques de seguretat per garantir que pot fer o no un client. Grups i usuaris poden ser definits per assignar permisos d'accés a diverses cues, tòpic o *connection factories*.

Respecte als canals de comunicació entre clients i servidors, també són un aspecte important quan es parla de seguretat. Un canal de comunicació pot ser assegurat mitjançant aïllament físic (com una xarxa dedicada) o amb l'encriptació de la comunicació entre el client i el servidor. Aquest últim implica un intercanvi de claus entre el client i el servidor. La clau permet al receptor descodificar i llegir el missatge. Actualment els proveïdors JMS solen incorporar SSL i encriptació del cos del missatge. SSL (*Secure Socket Layer*) és un estàndard per la comunicació segura utilitzada amb freqüència a Internet. Amb SSL, el protocol del proveïdor JMS és encriptat protegint cada aspecte amb l'intercanvi de missatges. L'encriptació del cos com a alternativa permet minimitzar el cost d'aquesta aplicant-la únicament als missatges que necessiten per la naturalesa de les dades que transporten, ser encriptats.

CAPÍTOL 4 Anàlisi de requeriments

4.1 DESCRIPCIÓ DEL PROJECTE

L'objectiu d'aquest capítol és formalitzar les diferents necessitats i requisits del projecte amb una major profunditat, per tal de definir els requeriments funcionals i no funcionals del sistema, els quals seran necessaris en la fase de disseny i implementació de l'aplicació.

Es pretén doncs, un monitor de servidors JMS que permeti facilitar tasques rutinàries als programadors i equips de proves com poden ser; la solució de problemes de connexió del clients connectats al servidors, si hi ha una sobrecàrrega de missatges acumulats sense consumir, l'enviament de missatges a destinacions, la consulta del contingut dels missatges emmagatzemats a una cua... de tal forma que esdevingui una eina útil i gratuïta per aquests. A més, el servidor de l'aplicació ha de poder executar-se en diverses plataformes.

L'equip de proves, necessita funcionalitats que permetin agilitzar les seves tasques. En primer lloc, serà necessària una opció d'enviament de missatges, tal com s'ha comentat anteriorment en aquest capítol. Normalment la comunicació entre components es realitza amb missatges XML, ja que permet la validació del missatge a partir d'un esquema compartit pel generador i el consumidor del missatge. És per això que per evitar possibles errades en el missatge a enviar, s'inclou la opció de validar un missatge a partir d'un esquema XSD[21]. Una segona funcionalitat seria la possibilitat d'esborrar missatges, per tal de que no es vagin acumulant missatges sense consumir en les destinacions, cal una opció per poder esborrar els missatges pendents en una cua concreta. Per últim, s'ha de poder examinar el contingut de missatges emmagatzemats en un servidor JMS però sense esborrar-los.

D'altra banda, tota aplicació requereix de traces per poder realitzar un seguiment de les activitats que es duen a terme, a més de tenir una eina per analitzar possibles errors d'implementació i facilitar la seva correcció. Per tant, és necessari un sistema de traces amb possibilitat de configurar nivells (*warning, debug, error, info*) i poder així filtrar el contingut d'aquestes.

Degut a les múltiples connexions que es poden realitzar amb diferents servidors JMS, és requereix la gestió i configuració d'aquestes de tal forma que no s'hagin d'introduir totes les dades necessàries per la connexió cada vegada que es vulgui accedir a un servidor JMS. L'usuari podrà guardar aquestes dades i recuperar-les en qualsevol moment.

21 **XSD:** és un llenguatge d'esquema utilitzat per descriure l'estructura i les restriccions dels continguts dels documents XML d'una forma molt precisa, més enllà de les normes sintàctiques imposades pel propi llenguatge XML.

4.2 REQUERIMENTS FUNCIONALS

4.2.1 Interfície gràfica

L'aplicació consistirà en una finestra principal des d'on es podran accedir als diferents mòduls i funcionalitats, juntament amb informació bàsica sobre les destinacions ja que és de gran utilitat i així s'evita que l'usuari hagi d'interactuar amb l'aplicació (a través de menús o finestres) per demanar aquests tipus de dades. La interfície gràfica contindrà doncs:

- **Finestra principal:** adjuntarà les funcionalitats i components descrits en aquest apartat. En funció del gestor de finestres que tingui l'usuari, l'aplicació prendrà l'aspecte definit en el sistema operatiu. La finestra principal contindrà una barra de menú, una barra d'eines i la descripció de les destinacions.

- **Barra de menú:** a través de la qual s'accediran a totes les funcionalitats del monitor; des de la configuració de paràmetres del servidor, fins la configuració de connexions, gràfiques, enviament de missatges, esborrament de missatges pendents i consulta del contingut de missatges.

- **Barra d'eines:** contindrà dreceres de les funcionalitats més rellevants de l'aplicació, com els botons de refresc o l'accés a la connexió d'un servidor JMS.

- **Informació de les destinacions:** tindrà una llista amb les destinacions creades al servidors dividides en funció de la seva naturalesa; cues, tòpics o durables, i es mostraran dades generals sobre aquestes, com els missatges pendents o la quantitat de missatges processats. A més, s'ha de poder escollir entre un refresc automàtic o manual de les dades. En aquest últim cas l'usuari ha de poder actualitzar les dades amb un botó situat a la finestra central. La informació a mostrar doncs, es compon de:

- Destinacions creades en el servidor JMS (cues, tòpics)
- Estadístiques de missatges: En funció del servidor (degut a les limitacions que tingui aquest per accedir a les seves dades), es poden llegir:
  - Missatges entrants i sortints (parcials i totals)
  - Missatges pendents

- Estadístiques generals del servidor: serien el total del conjunt de paràmetres monitoritzats per cada destinació. Ajuden a tenir una primera impressió sobre l'estat del servidor JMS.

4.2.2 Gestió de la configuració i connexió

a) Creació i configuració d'una sessió amb un servidor JMS.

L'usuari podrà crear sessions de connexions contra un servidor JMS de forma gràfica. Aquestes sessions es podran carregar, guardar o modificar tal com s'explica en el següents apartats.

Per tal d'establir una connexió amb un servidor JMS, s'haurà d'introduir el nom de la sessió, la direcció i port del servidor, el nom de usuari, contrasenya com a mínim. Les dades introduïdes seran emmagatzemades de forma persistent, a fi de recuperar-les en un futur.

b) Emmagatzemar les sessions:

El sistema guardarà en un fitxer en el directori d'instal·lació del monitor amb totes les configuracions que l'usuari ha introduït, per a la seva posterior recuperació.

c) Recuperar les sessions:

L'usuari podrà recuperar una configuració emmagatzemada per tal d'establir una sessió de connexió amb un servidor JMS.

d) Editar les sessions:

Es podran editar les sessions existents per tal de canviar alguns paràmetres a posteriori mitjançant sempre una interfície gràfica.

4.2.3 Dades mostrades

a) Cues, tòpics i durables.

Es mostrarà el nom de les destinacions creades en el servidor JMS (cues, tòpics i subscripcions durables) en la finestra principal de l'aplicació.

b) Estadístiques.

Per cada destinació es mostrarà informació diversa com pot ser els missatges que estan en el servidor per una destinació en concret, els missatges entrants i de sortida, etc. El nombre d'estadístiques disponibles dependrà del servidor JMS, ja que en funció de la seva API i mètodes d'accés al mateix, es podrà mostrar unes dades o unes altres. En general es mostraran els missatges entrants i sortints (*parcials calculats des de l'últim refresc de dades i totals*) i missatges pendents (*numero de missatges*) sempre que el servidor JMS tingui aquestes dades accessibles. S'hauran de mostrar en la finestra principal de l'aplicació.

c) Refresc de les dades.

Es podrà optar per un refresc automàtic de les dades cada segon o per un refresc manual forçat per l'usuari. D'aquesta forma es redueix el tràfic de missatges durant temps d'inactivitat per part de l'usuari amb el servidor JMS.

d) Gràfiques.

Serà possible obrir gràfiques sobre un conjunt d'estadístiques, per tal de monitoritzar més fàcilment les dades generals del servidor, amb possibilitat de copiar-les o exportar-les a un altre format. Les gràfiques disponibles seran:

- Total de missatges entrants en el servidor.
- Total de missatges de sortida en el servidor.
- Total de missatges entrants parcials (des de l'últim refresc).
- Total de missatges de sortida parcials (des de l'últim refresc).
- Missatges total pendents
- Número total de cues, tòpics i subscripcions durables.

4.2.4 Accions sobre el servidor

a) Enviar missatge.

Es permetrà l'enviament d'un missatge de tipus *TextMessage* especificat per l'usuari, a una destinació. Es podrà adjuntar un esquema XSD, per tal de validar el missatge en cas que sigui un XML. En aquesta fase del projecte només es tindrà en compte aquest tipus de missatges, però s'haurà de tenir en compte possibles ampliacions i permetre més tipus de missatge JMS per enviar.

b) Netejar destinació.

Habilitarà l'eliminació dels missatges pendents en una destinació en concret.

c) Consultar missatges d'una destinació

S'obtindran els missatges pendents en una cua, per tal de consultar el seu contingut, però sense consumir els mateixos.

d) Creació de destinacions

Sempre que l'usuari introduït per les tasques d'administració tingui permisos (definits òbviament al servidor JMS), es podran crear cues, tòpic i subscripcions durables.

e) Eliminar destinacions

Es permetrà eliminar cues, tòpics i subscripcions durables (sempre que es disposi de permisos d'administració en aquell servidor JMS).

4.3 REQUERIMENTS NO FUNCIONALS

A més de les funcionalitats explicades anteriorment, el monitor haurà de tenir en compte les següents consideracions.

En primer lloc, és prioritari minimitzar el tràfic de missatges. Cal que el disseny del monitor no saturi al servidor JMS amb peticions excessives de refresc de dades. A més, com s'ha dit en nombroses ocasions, la interfície ha de ser simple i fàcil d'utilitzar: l'aplicació GUI ha de ser intuïtiva i ha de mostrar les dades necessàries i fonamentals, si no, seria qüestionable la utilitat del present projecte per l'usuari final.

Un altre aspecte és la inclusió de traces en el monitor per a mostrar informació sobre els processos interns del monitor i ajudi a la depuració del codi. El sistema ha de generar traces que es puguin configurar en funció d'un nivell (depuració, informació, errors...) sense necessitat de recompilar el codi.

Per últim, cal destacar la capacitat d'independència de servidor JMS; el monitor ha de ser independent a la implementació de la lògica d'administració d'un servidor JMS, assegurant la compatibilitat amb qualsevol proveïdor sense fer canvis en el codi del monitor.

CAPÍTOL 5 Disseny de l'aplicació

5.1 CONFIGURACIÓ DE LA PLATAFORMA

En aquest apartat es resumiran totes aquelles decisions i detalls sobre els aspectes de l'entorn de desenvolupament i proves que afecten al present projecte, des de els sistemes operatius fins les llibreries emprades en la implementació de l'aplicació.

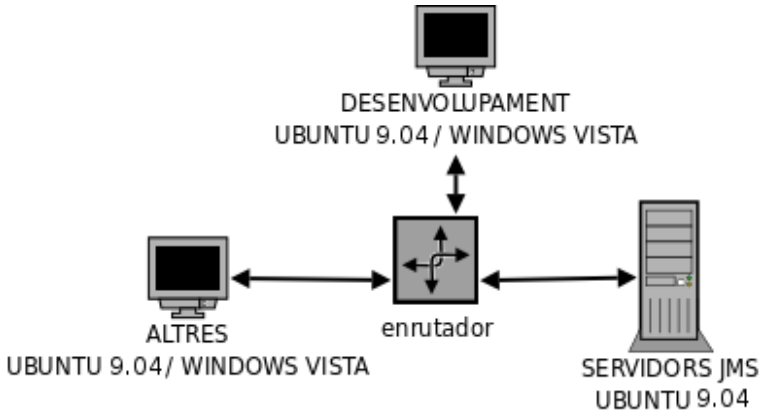


Figura 19: entorn de desenvolupament i proves del projecte

5.1.1 Sistema operatiu.

En el present projecte es faran servir dos sistemes operatius, per provar que efectivament el monitor és multiplataforma i ajudar a pal·liar possibles incompatibilitats de l'aplicació en diferents sistemes. Els sistemes operatius escollits són *Windows Vista Home Edition* i *Windows XP Profesional* de *Microsoft* i *Ubuntu* 9.04. de *Canonical*. L'elecció de Vista es basa en un motiu econòmic; es disposaven d'una llicència pel seu ús, malgrat que el consum de recursos d'aquest és elevat, serà útil per realitzar proves del sistema a implementar en altra plataforma que no sigui Linux. Windows XP s'utilitzarà a més de les proves, en la gestió del projecte, ja que s'utilitza *Microsoft Project* 2003. Per la resta del projecte, Ubuntu 9.04 serà l'utilitzat per tasques de desenvolupament, elaboració de la memòria, proves i fins i tot la realització de diagrames mitjançant l'aplicació DIA. L'elecció de Ubuntu es recolza en la facilitat d'ús i l'alta compatibilitat amb diferents tipus de maquinari, sent una distribució *Linux* amigable per l'usuari, reduïnt així problemes i per tant, dedicant menys temps a la configuració del propi sistema en favor del present projecte. Ubuntu es pot descarregar gratuïtament de la pàgina oficial i està disponible per diferents tipus de processadors.

Serà recomanable que es disposi en els ordinadors que es dediquin al desenvolupament, una partició amb els dos sistemes operatius. Així doncs es procedirà a instal·lar primer el sistema de *Microsoft*, ja que Ubuntu instal·larà per defecte un gestor d'arrancada anomenat GRUB, que permetrà escollir entre ambdós sistemes al arrencar la computadora.

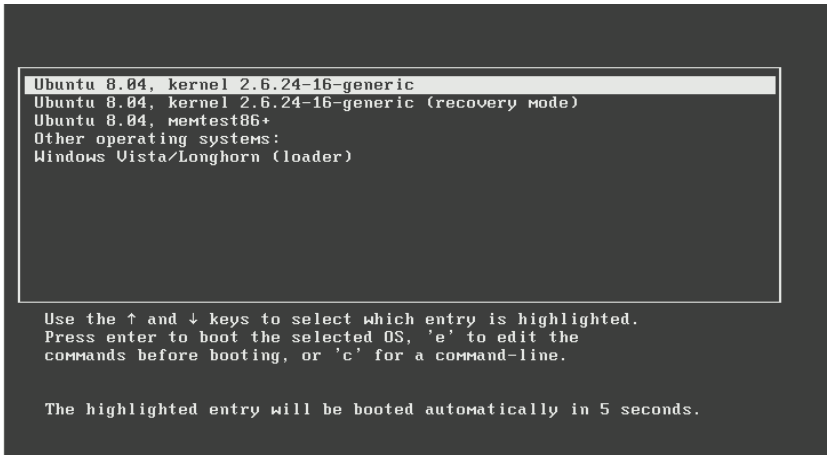


Figura 20: captura de pantalla del gestor d'arrancada GRUB

5.1.2 Servidors JMS

Al projecte es treballarà amb dos proveïdors de missatgeria JMS. Es recomanable tenir una màquina dedicada on instal·lar aquest servidors, accessible a tots els equips de desenvolupament per tal d'alliberar els recursos d'aquestes maquines dedicades a la implementació del projecte. No obstant, les llibreries que subministren aquests pel desenvolupament d'aplicacions, s'hauran de copiar a tots els equips destinats a implementar l'aplicació. Cal dir, que s'ha de garantir que els usuaris d'aquestes aplicacions tinguin permisos per poder executar-les.

OPENJMS

El proveïdor de missatgeria OpenJMS és pot descarregar a la següent direcció: http://openjms.sourceforge.net/downloads.html

El paquet no conté cap instal·lable. Es descomprimeix i es copia directament al directori on pengen les aplicacions de l'usuari. L'execució del servidor es realitza mitjançant un *script* i no cal compilar-lo. El servidor arranca amb la següent instrucció, dintre del directori bin de l'aplicació:

```
nohup ./openjms.sh start &
```

Per parar-lo:

```
./openjms.sh stop
```

FUSE

Caldrà registrar-se prèviament per descarregar el producte. Una vegada es té un compte d'usuari, es pot baixar la versió Linux, Mac o Windows de la següent direcció: http://fusesource.com/downloads/ . Es tracta d'un instal·lable amb interfície gràfica:

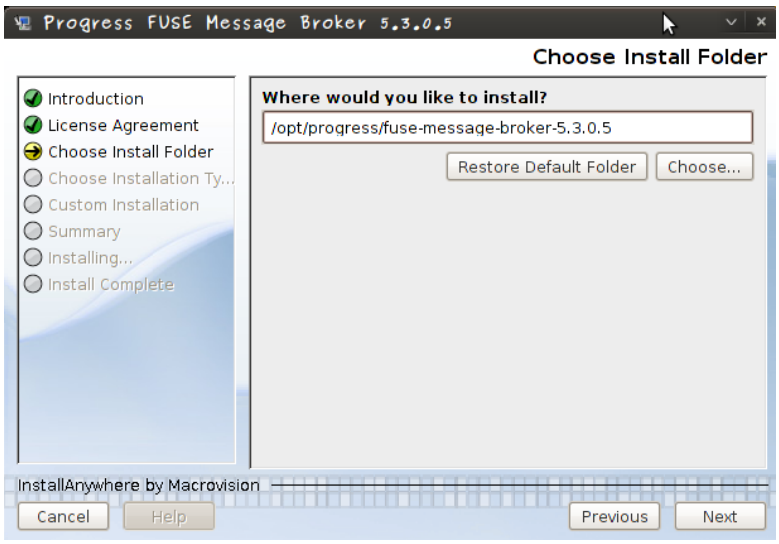


Figura 21: instal·lació de Fuse

Per arrancar el servidor s'utilitzarà la següent instrucció dintre del directori bin de l'aplicació:

```
nohup ./activemq &
```

Per parar el servidor es pot matar el procés:

```
ps -ef | grep activemq

kill [PID]
```

On [PID] és l'identificador del procés imprès pel comand “ps”.

Per provar que efectivament el servidor està funcionant correctament, s'obre un navegador i s'introdueix la següent URL http://localhost:8161/admin/ . Cal dir que *Fuse* és una implementació de *ActiveMQ,* per tant l'estructura de directoris de l'aplicació és semblant a la d'aquest.

5.1.3 Entorn desenvolupament integrat.

Tal com ja es va dir a l'anàlisi de viabilitat, Eclipse és el entorn de desenvolupament escollit. Accedint a la pàgina web oficial (http://www.eclipse.org/downloads/) es pot descarregar l'última versió disponible de aquest IDE disponible per plataformes *Windows, Linux* i *Mac*.

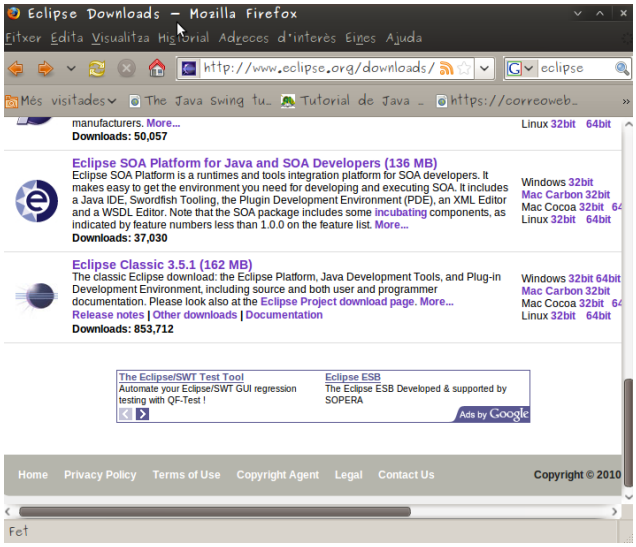


Figura 22: descàrrega de Eclipse Classic

Pel projecte actual, s'opta per baixar la versió clàssica de Eclipse, ja que no seran necessàries eines per desenvolupaments web o *Java EE*.

A més de l'entorn de desenvolupament integrat, serà necessari obtenir el Java Development Kit (JDK) per tal de poder compilar el codi. Es pot descarregar de la pàgina oficial de *Sun Microsystems* (http://java.sun.com/javase/downloads/widget/jdk6.jsp) , i existeixen versions per plataformes *Windows, Linux, Mac* i *Unix*. Pot ser interessant tenir varies JDK instal·lades per a

compilar amb diferents versions d'aquesta. Per tant, s'instal·laran a un directori les diferents versions de JDK amb que es vulguin treballar, i es crearà un enllaç simbòlic a la versió actual. Així doncs, es podrà indicar al IDE, quina versió de JDK utilitzarem per compilar i executar el codi implementat.

5.1.4 Llibreries.

A més de les llibreries que s'inclouen en els proveïdors de JMS emprats en el projecte, pel desenvolupament s'utilitzarà una llibreria per la generació de gràfiques (*JFreecharts*) i una per *logging* de l'aplicació (Log4j). Ambdues llibreries són gratuïtes i es poden descarregar de les seves pàgines web corresponents:

- http://www.jfree.org/jfreechart/download.html
- http://logging.apache.org/log4j/1.2/download.html

Per tant, s'inclouran dintre del *classpath*[22] de l'aplicació les llibreries *log4j-1.2.15.jar, jcommon-1.0.16.jar* i *jfreechart-1.0.13.jar .*

5.1.5 Altres aplicacions.

MICROSOFT PROJECT 2003

Microsoft Project és un programari d'administració de projectes dissenyat, desenvolupat i comercialitzat per *Microsoft* per assistir a administradors de projectes en el desenvolupament de plans, assignació de recursos a tasques, donar seguiment al progrés, administrar pressupost i

22 ***Classpath***: és el conjunt de rutes on es troben les llibreries i classes a l'hora d'executar el programa.

analitzar càrregues de treball. Aquest serà instal·lat a les màquines que disposin d'un dels dos sistemes operatius *Microsoft* disponibles a l'entorn de desenvolupament del projecte (Vista / XP).

OPENOFFICE 3.0

OpenOffice.org és un projecte comunitari per crear una paquet ofimàtic basat en codi obert (amb llicència LGPL), procedent d'una versió antiga de *Star Office* de *Sun Microsystems*. Aquest serà instal·lat en totes les màquines i particions disponibles en el projecte, ja que serà necessari per la redacció de documentació i de la present memòria. OpenOffice es pot descarregar de la pàgina oficial del projecte (http://download.openoffice.org).

DIA

Dia és una aplicació informàtica per a la creació de diagrames, desenvolupada com a part del projecte GNOME. Està dissenyat com un substitut de l'aplicació comercial *Visio* de *Microsoft*. Es pot utilitzar per dibuixar diferents tipus de diagrames. Actualment s'inclouen diagrames entitat-relació, diagrames UML, diagrames de flux, diagrames de xarxes, diagrames de circuits elèctrics, etc. S'instal·larà en totes les particions amb Ubuntu per la creació de diagrames en la documentació del projecte.

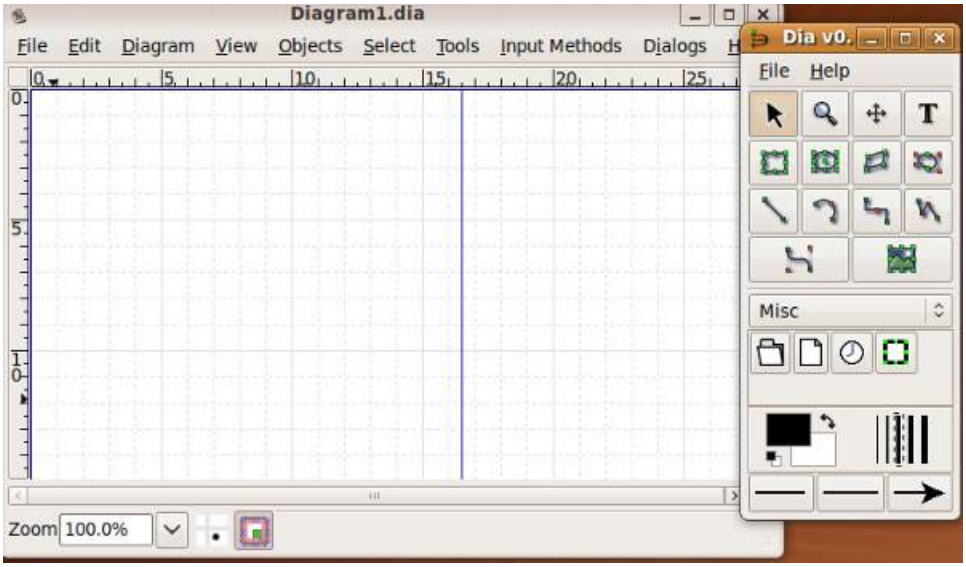


Figura 23: aplicació DIA

5.2 CASOS D'ÚS

L'objectiu és tenir una visió a alt nivell del comportament de l'aplicació i per això és convenient realitzar un diagrama de casos d'ús:

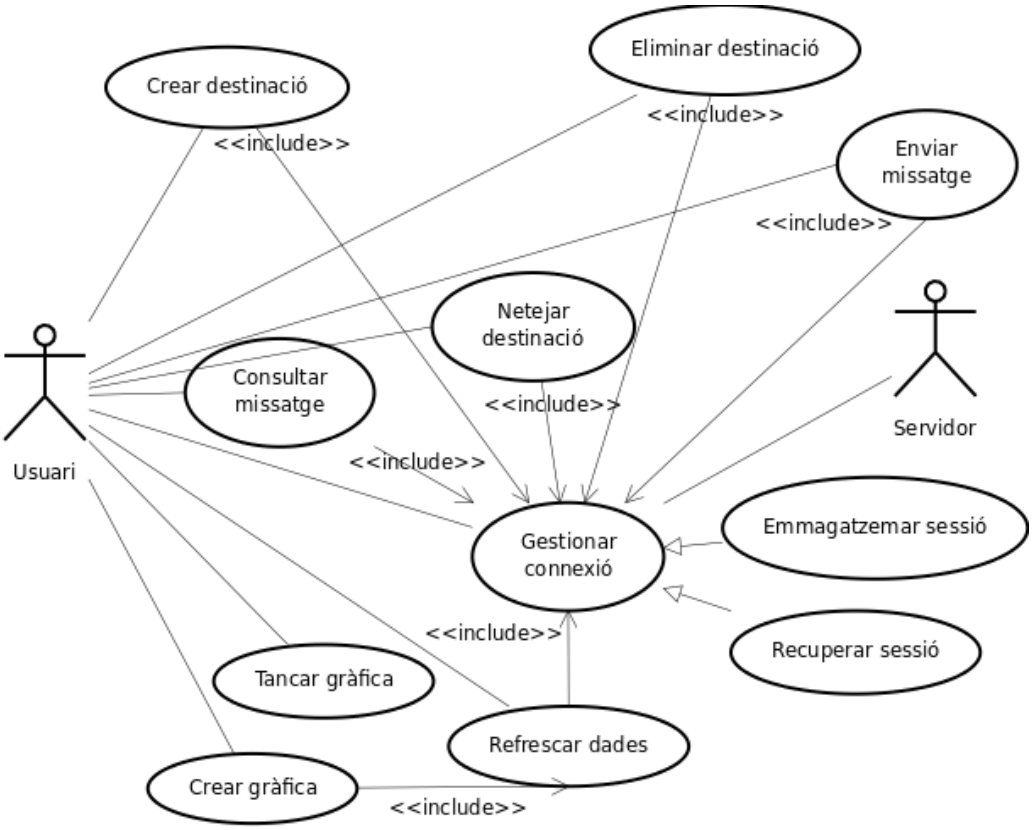


Figura 24: diagrama de casos d'ús de JMSMonitor

Tot seguit es resumiran els casos d'ús que descriuen el sistema a desenvolupar en el present projecte.

5.2.1 Gestionar connexió

Aquest cas d'ús té com a finalitat l'administració de la connexió amb els diferents servidors JMS. Mitjançant unes dades facilitades per l'usuari, intentarà establir contacte amb el servidor per tal de que es puguin obtenir les dades sol·licitades pel monitor. En cas d'error es mostrarà a l'usuari la causa de l'excepció.

Aquesta funcionalitat es podrà activar mitjançant el menú principal de l'aplicació i carregarà en el sistema les dades necessàries per poder establir una connexió. Cal dir que l'usuari podrà forçar la connexió i desconnexió d'una mateixa sessió, directament en la interfície principal de la aplicació mitjançant un botó després de ser carregada.

L'usuari haurà de complimentar la següent informació:

Paràmetre	Descripció
Nom de la sessió	Nom que dona l'usuari a la sessió de connexió a guardar en el monitor.
Direcció de administració	URL del servidor JMS i port habilitada per l'administració del servidor.
Usuari	Nom de l'usuari del servidor JMS(recomanable que tingui permisos d'administració per tal d'assegurar la màxima funcionalitat del monitor).
Clau	Contrasenya de l'usuari.
Factory	Nom JNDI d'una *factory* del servidor per tal de realitzar accions (enviament i consulta de missatges) sobre el servidor JMS.
Direcció de connexió	Direcció i port del servidor per poder rebre i consultar missatges.
Tipus de servidor JMS	Proveïdors de JMS (OpenJMS, ActiveMQ...)
Llibreries d'implementació del proveïdors	Ubicació de la llibreria proporcionada pel monitor, que implementa els mètodes d'obtenció de dades per aquell proveïdor de JMS.
Temps de refresc de dades	Interval de refrescs automàtics de dades en mili-segons.

Taula 15: paràmetres del gestor de sessions

Com es pot veure al diagrama de casos d'ús en l'apartat anterior, hi ha dos casos d'ús que completaran la funcionalitat; emmagatzemar i recuperar sessió. Aquests complementen la funcionalitat bàsica afegint la possibilitat de guardar les dades introduïdes per l'usuari, carregar-les de nou i canviar-les a posteriori.

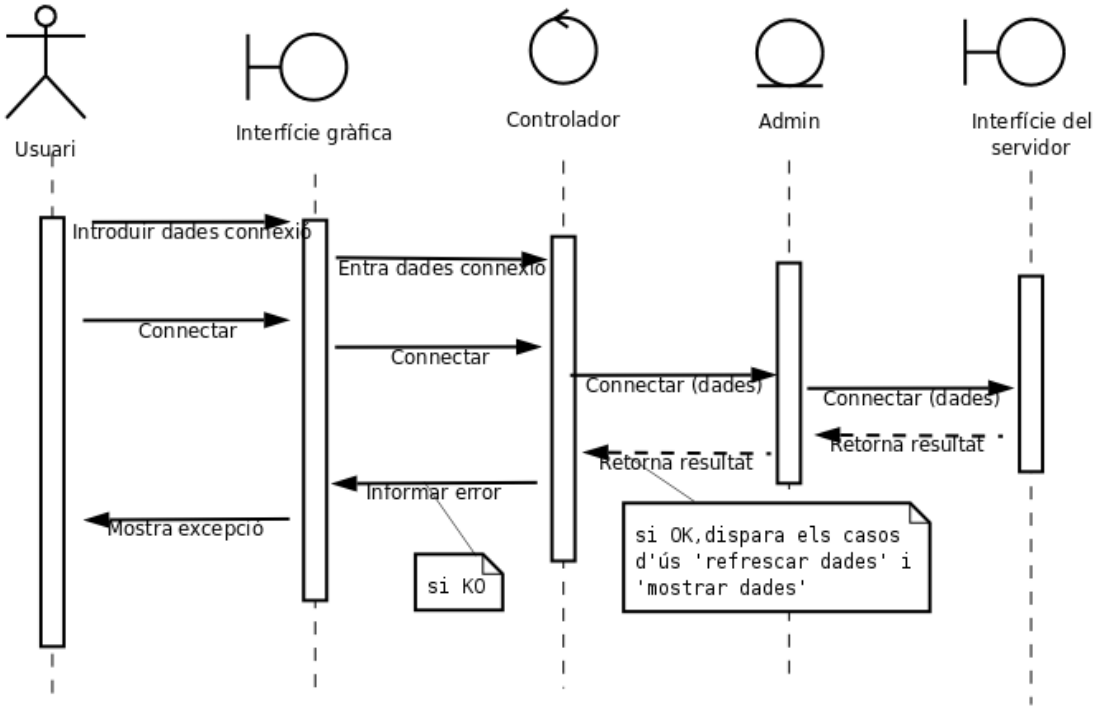


Figura 25: diagrama de seqüència de la connexió amb un servidor JMS

En el cas que s'estableixi una connexió amb èxit amb el servidor JMS s'executen dos casos d'ús, “refrescar dades” i “mostrar dades”, ja que l'usuari no demanarà explícitament les dades estadístiques del servidor si no que el monitor anirà refrescant cada cert temps la vista amb les dades actualitzades.

5.2.2 Emmagatzemar sessió

S'encarrega de guardar en un fitxer les dades introduïdes per l'usuari referents a la connexió amb el servidor JMS per poder-les recuperar més endavant. Una vegada l'usuari a introduït les dades en el cas d'ús anterior, podrà escollir emmagatzemar-les. Aquesta funcionalitat serà accessible mitjançant la interfície de la gestió de connexions; on es disposà d'un botó per tal de poder salvar la sessió.

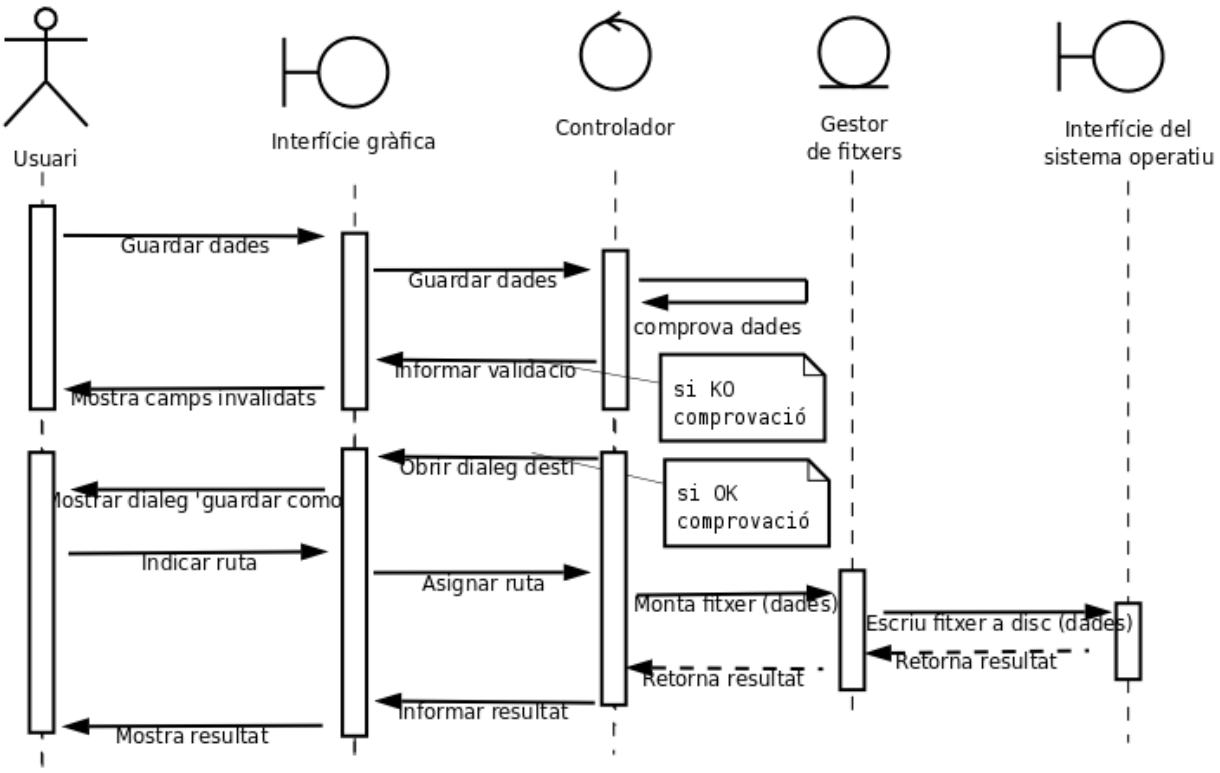


Figura 26: diagrama de seqüència de "guardar dades"

Les dades candidates a emmagatzemar seran comprovades i validades per assegurar de que tots els camps obligatoris estan correctament informats. En cas negatiu, s'informarà a l'usuari dels camps que falten. Si la comprovació és positiva, el sistema mostrarà a l'usuari quin nom tindrà el fitxer i on es guardarà, per poder crear l'arxiu i escriure'l a disc.

5.2.3 Recuperar sessió

Habilita a l'usuari la possibilitat d'obrir les dades que va emmagatzemar en l'anterior cas d'ús. Una vegada ha recuperat les dades de la sessió, l'usuari podrà editar-les i emmagatzemar-les de nou (veure 'emmagatzemar sessió') o establir una connexió (veure 'gestionar connexió'). Aquesta funcionalitat serà accessible mitjançant la interfície de la gestió de connexions; on es disposà d'un botó per tal de poder recuperar la sessió.

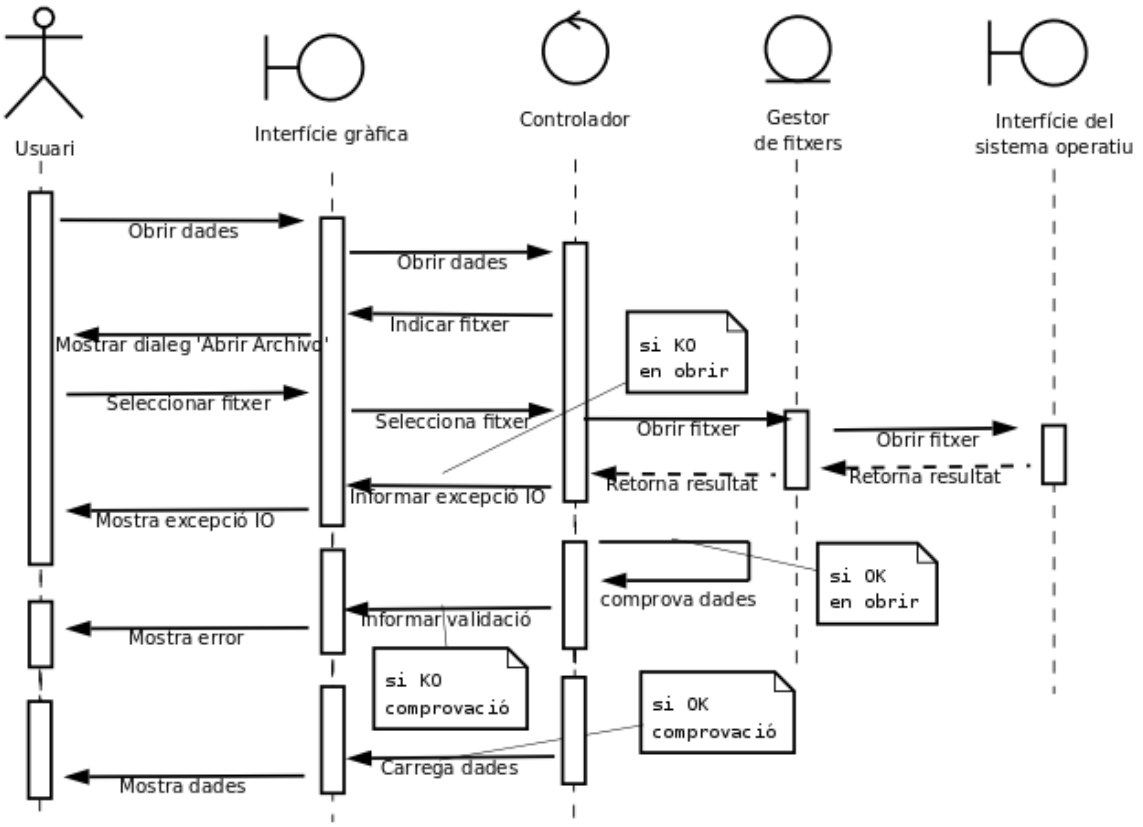


Figura 27: diagrama de seqüència de recuperar sessió

L'usuari seleccionarà l'opció de recuperar sessió i escollirà un fitxer mitjançant un diàleg. Una vegada accepta, el sistema intentarà obrir el fitxer indicat. En cas d'excepció d'entrada i sortida, es mostrarà a l'usuari i el cas d'ús finalitzarà. Si el procés de càrrega es correcte, el sistema verificarà la integritat del fitxer i en cas positiu, mostrarà les dades a l'usuari per a que pugui realitzar una acció amb aquestes (establir connexió o editar-les), en cas contrari, informarà a l'usuari de que el fitxer no era vàlid.

5.2.4 Refrescar dades

El present cas d'ús s'encarrega de actualitzar les dades mostrades en el monitor. El refresc de l'aplicació per defecte serà automàtic, però l'usuari pot desactivar aquesta opció mitjançant un botó a la interfície principal de l'aplicació. L'usuari podrà refrescar de forma manual les dades amb un altre botó situat a la interfície principal.

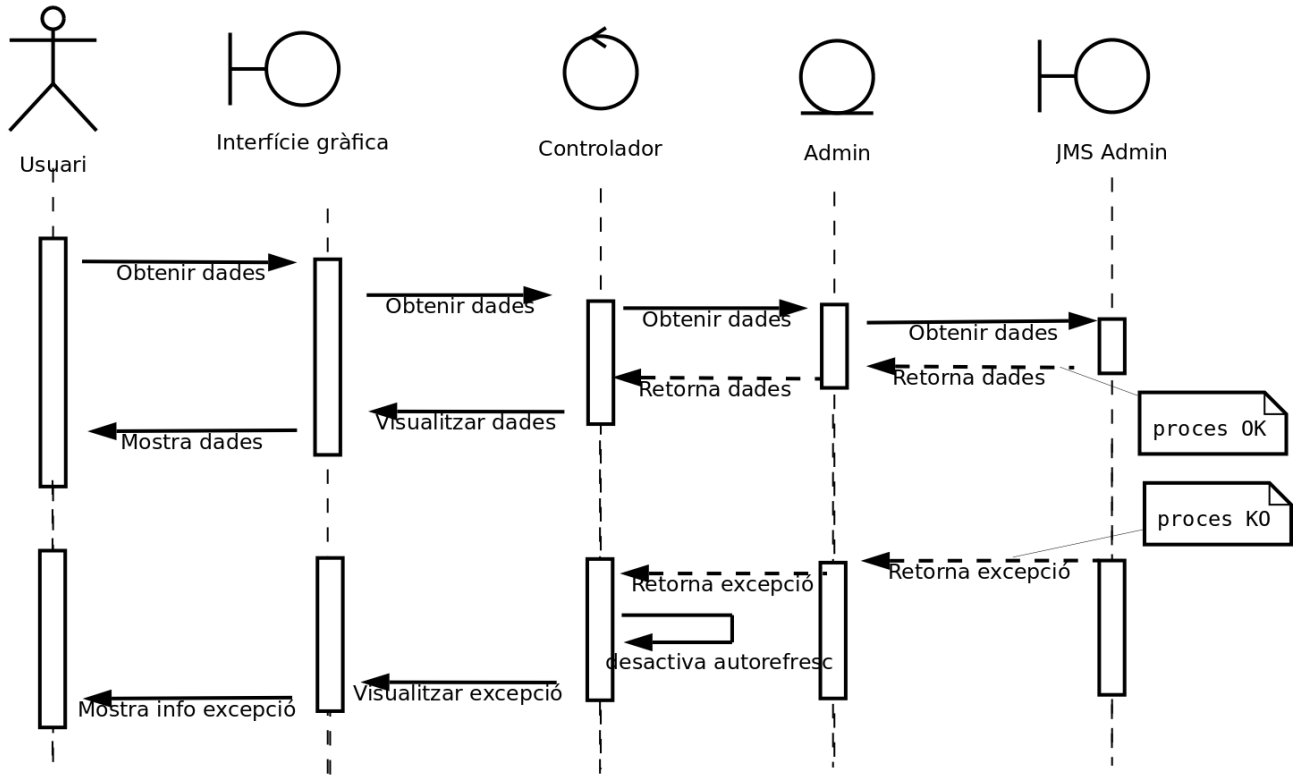


Figura 28: diagrama de seqüència de "refrescar dades"

L'usuari (o el procés d'autorefresc) demanarà l'actualització de les dades a l'administrador connectat amb el servidor. Les dades una vegada retornades per aquest, es mostraran de nou en la interfície principal de l'aplicació. En cas que s'aixequés una excepció durant el procés d'obtenció de les dades, el sistema mostrarà l'excepció a l'usuari i desactivarà l'autorefresc per evitar la continua creació de missatges d'excepció ja que podria ser que la connexió s'hagi caigut o hi hagi problemes amb la xarxa.

5.2.5 Enviar missatges

Aquest cas d'ús permet a l'usuari la creació i l'enviament de missatges de tipus *TextMessage.* Així doncs, l'usuari mitjançant el menú de l'aplicació o el menú contextual de la destinació seleccionada accedirà a omplir un formulari per tal de crear i enviar un missatge. Haurà d'emplenar els següents camps:

Concepte	Descripció	Tipus d'objecte	Obligatori a omplir per l'usuari	Valor per defecte
Dades bàsiques				
Destinació	Cua o tòpic on s'enviarà el missatge	Llista desplegable	SI	Si s'ha accedit mitjançant el menú contextual de la destinació seleccionada serà ella mateixa, si no cap.
XML	S'indicarà la ruta del XSD a adjuntar per validar el cos del missatge (en cas de que es vulgui enviar un XML)	Cercador de fitxer	NO	N/D
Enviar	Confirmar la creació i l'enviament del missatge	Botó	N/D	N/D
Cancel·lar	Cancel·lar la creació del missatge	Botó	N/D	N/D
Cos del missatge				
Contingut	El contingut del missatge	Àrea de text	NO	buit

Taula 16: camps per l'enviament de missatges

Per tal de que el codi sigui independent de les funcions que proporciona cadascuna de les distintes llibreries dels proveïdors de JMS, es buscarà el nom de la *factory* realitzant un "*lookup*" del JNDI d'aquesta al server, amb la finalitat de crear una sessió que permeti establir una connexió del productor que crearà i enviarà el missatge a la destinació indicada per l'usuari. Es recorda que la *factory* es defineix al crear una sessió de connexió del monitor.

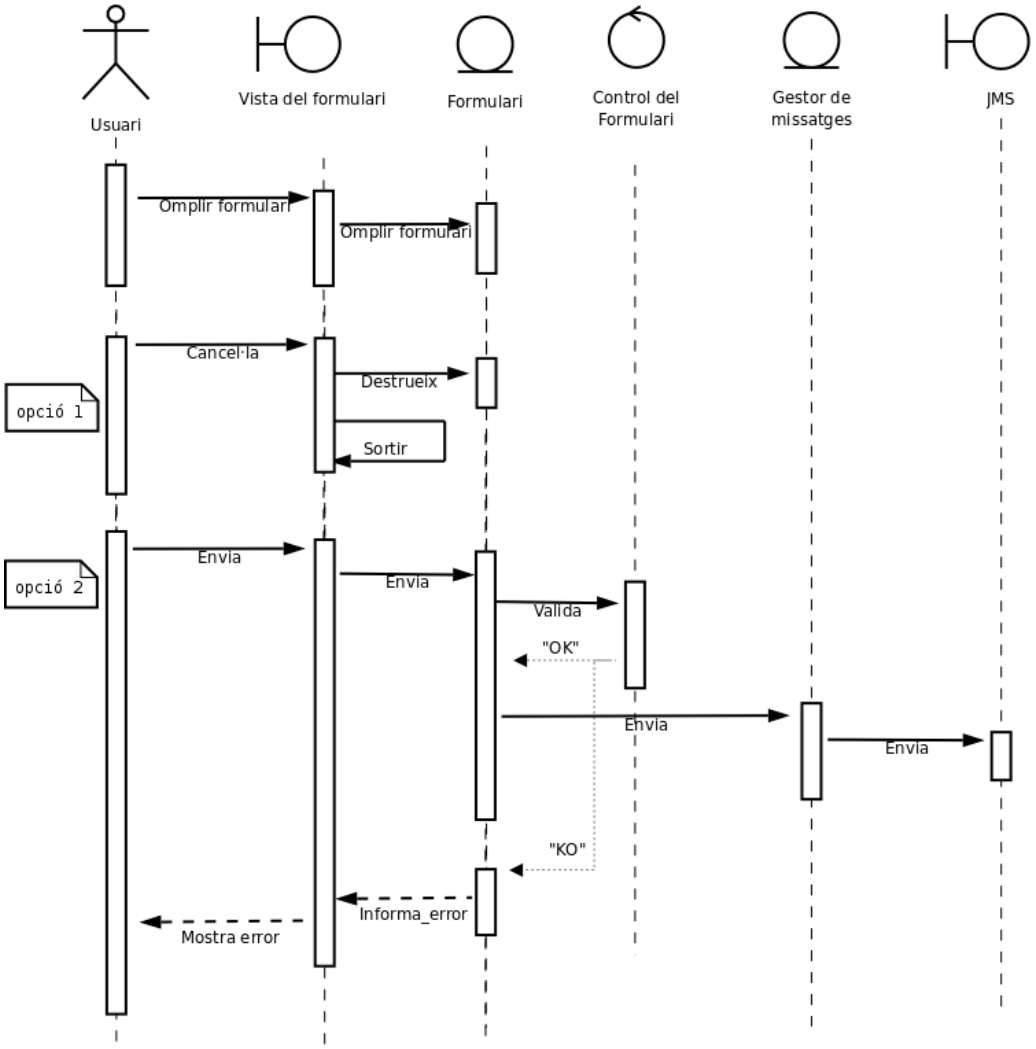


Figura 29: diagrama de seqüència de "enviar missatge"

5.2.6 Netejar destinació

Permetrà a l'usuari netejar tots els missatges que es troben a una cua pendents de consumir-se. Es podrà accedir mitjançant el menú principal de l'aplicació o amb el menú contextual de la destinació seleccionada (només per cues i durables). Abans però de realitzar la neteja, es demanarà la confirmació de l'usuari.

El client utilitzarà la interfície d'administració per gestionar l'esborrament dels missatges, ja que ni han proveïdors que proporcionen funcionalitats per aquest tipus de tasques. En cas que no existeixi cap mètode, s'haurà d'implementar dintre del connector que uneix el client amb el servidor de JMS.

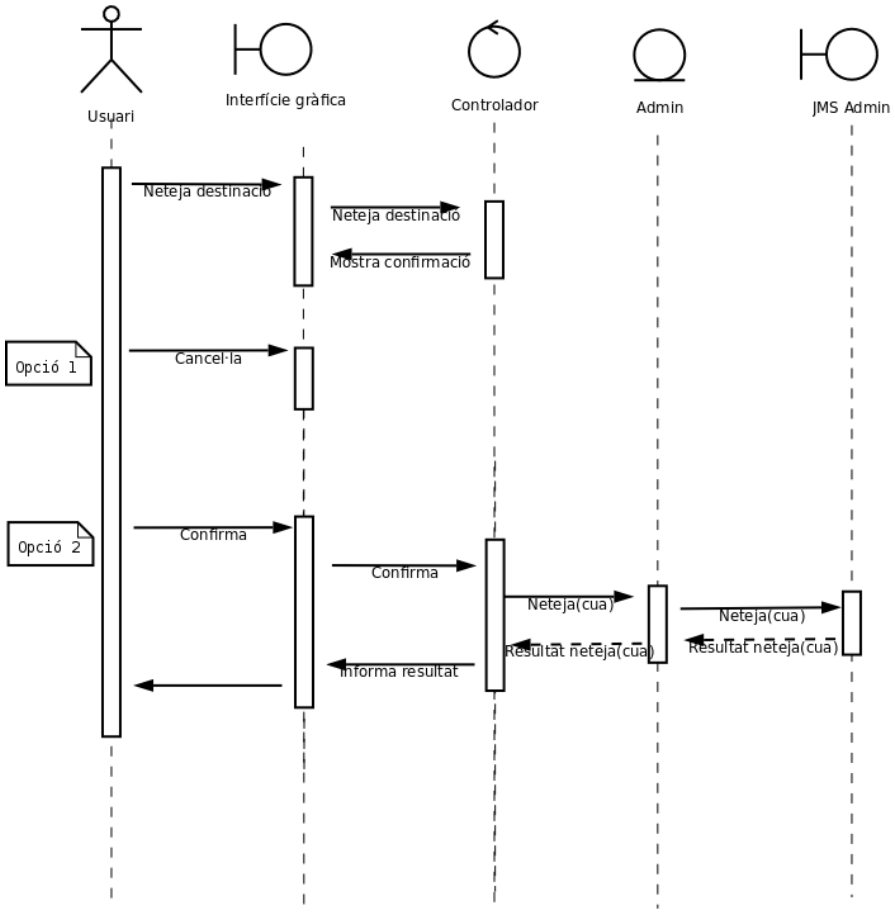


Figura 30: diagrama de seqüència de "esborrar missatges"

5.2.7 Consulta missatges de la destinació

Aquest cas d'ús habilita la possibilitat a l'usuari de consultar el contingut dels missatges emmagatzemats a una cua sense consumir-los de la mateixa, és a dir, no desapareixen del servidor en el moment de ser llegits. Serà accessible mitjançant el menú contextual de la destinació o fent un doble *click* a la destinació objectiu. La informació que mostrarà serà tant la capçalera com el cos del missatge.

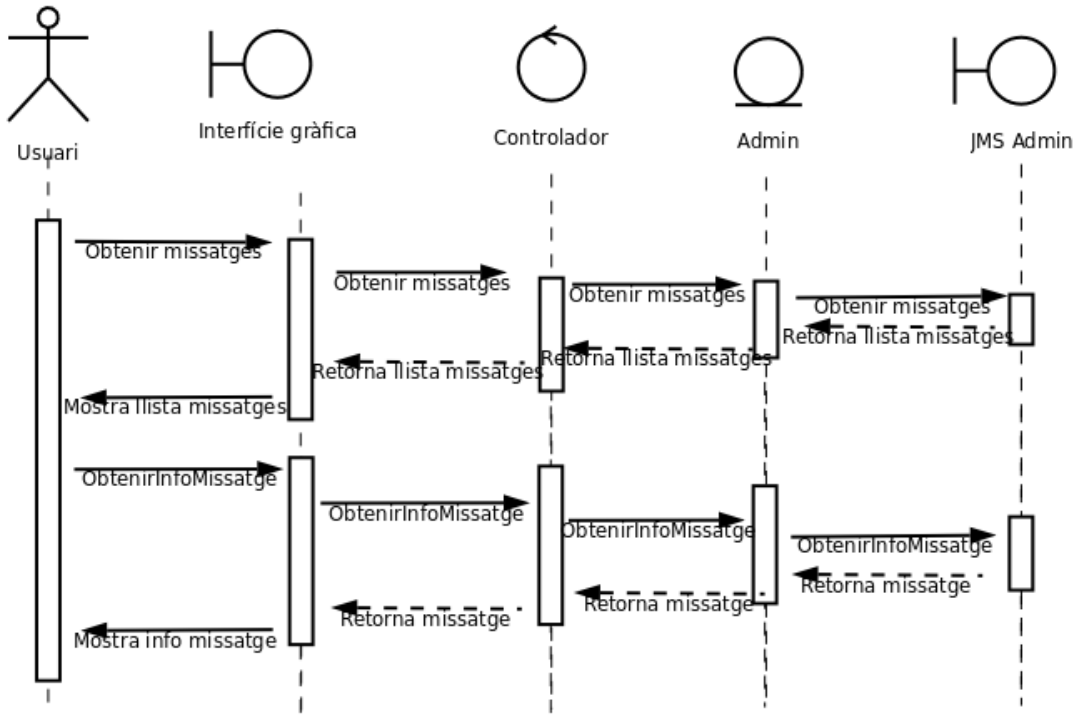


Figura 31: diagrama de seqüència de "obtenir missatges"

L'usuari primer sol·licitarà que el monitor mostri una llista amb els missatges que n'hi han al destí sol·licitat. Una vegada es mostra i l'usuari selecciona un missatge en concret, el sistema obtindrà la informació d'aquest i la visualitzarà.

5.2.8 Crear gràfica

Permet a l'usuari crear gràfiques sobre variables del servidor JMS. Aquests paràmetres en ser actualitzats pel cas d'ús “refrescar dades”, forçaran també l'actualització de les gràfiques. La funcionalitat serà accessible mitjançant el menú principal de l'aplicació el tipus de gràfica seleccionat.

En el següent diagramà de seqüència, s'observa com l'usuari demanarà al sistema per la creació d'una gràfica en concret. El sistema crearà la gràfica i la situarà al front de la vista.

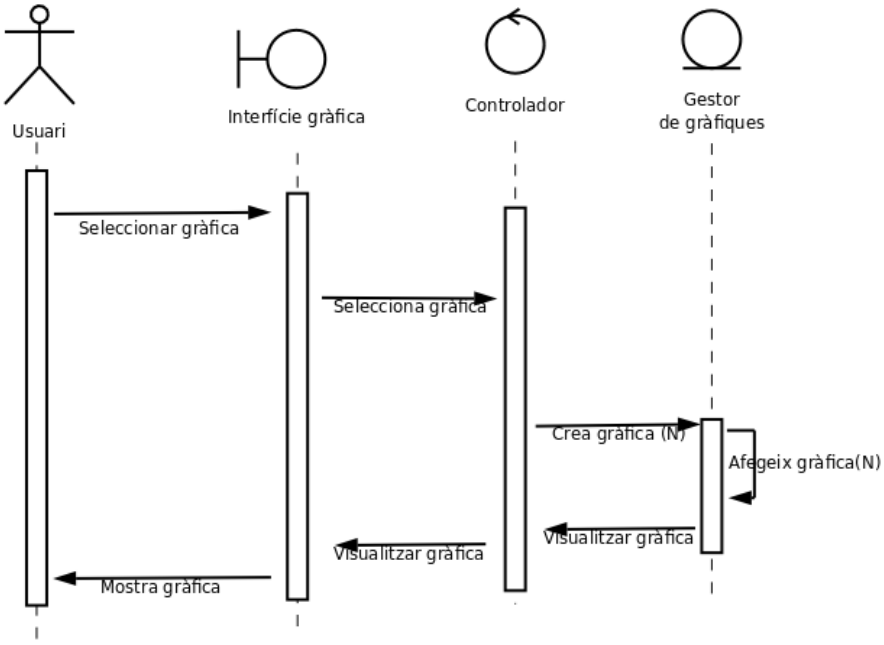


Figura 32: diagrama de seqüència de "crear gràfica"

5.2.9 Tancar gràfica

Aquest cas d'ús elimina una gràfica en el monitor. L'usuari únicament tindrà que tancar la finestra creada on es mostrà la gràfica. Aquesta acció farà que s'actualitzi la llista on es guarda la referència de les gràfiques eliminant aquella que s'ha donat de baixa per tal de no intentar actualitzar-la de nou.

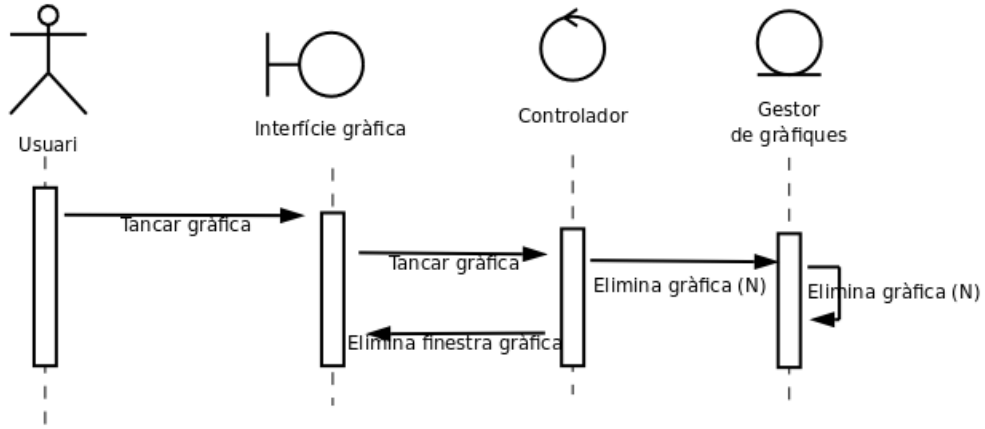


Figura 33: diagrama de seqüència de tancar gràfica

5.3 ALTRES CONSIDERACIONS DEL DISSENY

5.3.1 Interfície amb proveïdors

Un dels aspectes més crítics del projecte sens dubte, és assegurar la compatibilitat del monitor amb els diferents servidors JMS, de manera que encara que en aquesta primera versió només s'entregui la implementació de connectors per a dos servidors, el disseny de l'aplicació faciliti el desenvolupament i la integració futura de més proveïdors JMS.

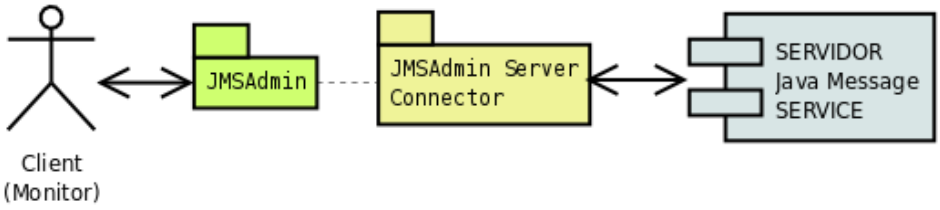


Figura 34: connexió del monitor amb el proveïdors de JMS

El problema principal resideix en que els objectes que necessiten de la informació dels servidors no poden utilitzar explícitament els mètodes i classes específics que subministra cada proveïdor de JMS, ja que el codi no seria genèric i qualsevol canvi de proveïdors JMS afectaria tant a aquelles classes que realitzessin operacions amb aquestes dades com a aquelles que representin la informació a l'usuari.

Una de les solucions seria desacoblar la part que adapta els mètodes específics de les API que proporcionen els proveïdors del codi del client. Amb el patró de disseny “Adapter”, es pot acomplir aquesta tasca tal com es mostra en el següent diagrama:

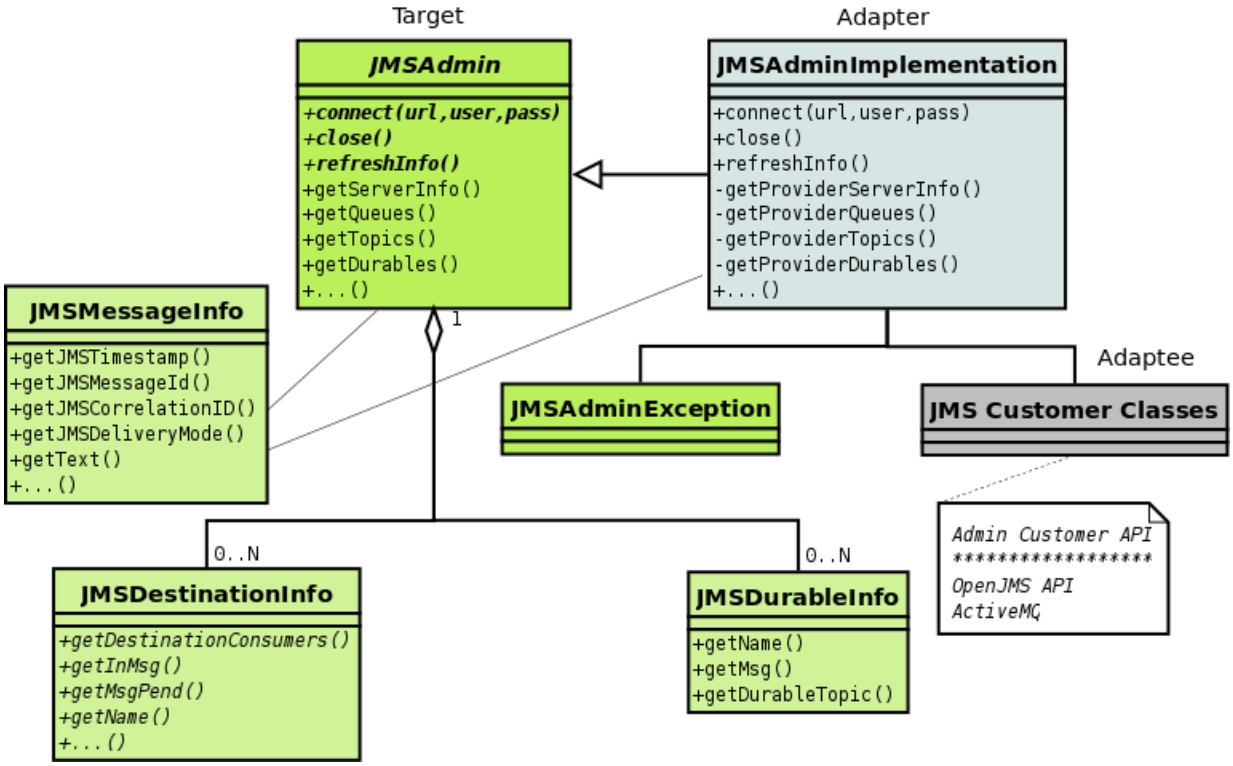


Figura 35: diagrama de classes amb el patró de disseny Adapter

El client utilitzarà l'abstracció *JMSAdmin* per tal d'accedir mitjançant la classe connectora (*JMSAdminImplementation*) als mètodes de la API d'administració del servidor JMS a monitoritzar.

Encara que l'objectiu del *adapter* és connectar dues parts que per si són incompatibles, per maximitzar la independència entre el client i la implementació de les classes adaptadores, aquelles que com s'ha comentat anteriorment s'encarreguen de connectar les API del proveïdors, es farà servir el patró de disseny “Bridge”. Aquest patró de disseny és recomanat quan es vol realitzar tasques com:

- Evitar un enllaç permanent entre l'abstracció i la seva implementació. Això pot ser degut a que la implementació ha de ser seleccionada o canviada en temps d'execució.

- Canvis en la implementació d'una abstracció no han impactar en els clients, és a dir, el codi no s'ha de recompilar.

5.3.2 Interfície gràfica

Amb l'objectiu de garantir la independència entre la vista i el model de dades i facilitar el desenvolupament de millores i ampliacions de l'eina, es decideix utilitzar les llibreries Swing de Java. Swing no té una arquitectura estrictament MVC[23] (el controlador i la vista es col·lapsen en una entitat), però permet separar la vista del model de dades. La API Swing de Java aporta:

- Àmplia varietat de components: Existeixen una amplia gama de objectes gràfics com botons, taules, menús... a més de contenidors per aquests.

- Aspecte modificable: Es pot personalitzar l'aspecte de les interfícies o utilitzar diversos aspectes que hi ha per defecte.

- Desacoblament de la vista i el model: Donant lloc a tot un enfocament de desenvolupament molt arrelat en els entorns gràfics d'usuari realitzats amb tècniques orientades a objectes. Cada element té associat una classe de model de dades i una interfície que utilitza. Es pot crear un model de dades personalitzat per a cada component, amb només heretar de la classe Model.

- Contenidors niats: Qualsevol component pot estar inclòs en un altre. Per exemple, un gràfic es pot niar en una llista.

- Diàlegs personalitzats: Es poden crear multitud de formes de missatges i opcions de diàleg amb l'usuari, mitjançant la classe *JoptionPane*.

- Classes per diàlegs habituals: Es pot utilitzar *JFileChooser* per triar un fitxer, i *JColorChooser* per triar un color.

23 ***MVC***: *Model-View-Controller* és un patró de disseny per al desenvolupament de programari que separa el model de dades, la interfície usuari i la lògica de control.

- Components per taules i arbres de dades: Mitjançant les classes *JTable* i *Jtree*.

- Potents manipuladors de text: A més de camps i àrees de text, es presenten camps de sintaxi oculta *JPassword*, i text amb múltiples fonts *JTextPane*. A més hi ha paquets per utilitzar fitxers en format HTML o RTF.

Existeixen nombrosos marcs de desenvolupaments MVC per treballar amb Java, però la corba d'aprenentatge d'aquests és alta i incrementa el nombre d'hores destinats pel projecte i per tant, serà menys costós utilitzar la llibreria de Swing per la representació de la informació que permet també separar el model de dades de la vista.

5.3.3 Gestió d'excepcions

Amb la finalitat d'unificar els tipus d'excepcions de diferents servidors, es crearan dintre de la *package* d'administració (que és la que connectarà el client monitor amb el servidor JMS), una serie de classes que serviran per separar el codi del connector del servidor JMS amb el client gràfic.

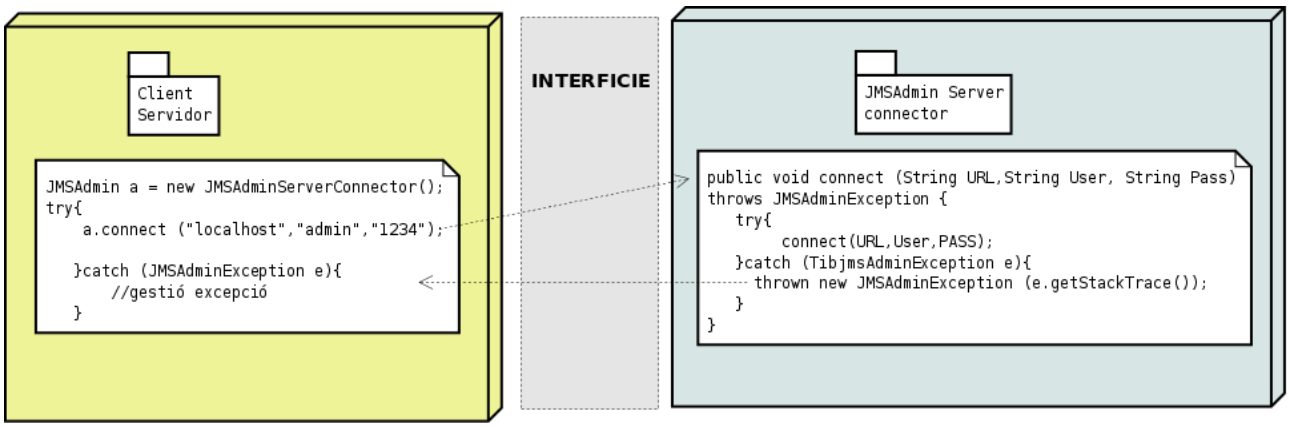


Figura 36: exemple de trucada del client GUI al connector.

En el següent exemple, es mostra la implementació de la funció de connexió per servidors TIBCO (no alliberat en el present projecte per qüestions de llicencies privatives). Com es pot veure en la declaració, la classe que utilitzi aquest mètode, haurà de gestionar l'excepció en cas que s'aixequi:

```
// IMPLEMENTACIO DE LA CONNEXIO DEL CONNECTOR PEL JMS SERVER TIBCO

public void connect(String URL, String USER, String PASSWORD)throws
                    JMSAdminException
{
   try{
         admin = new TibjmsAdmin(URL,USER,PASSWORD);
      } catch (TibjmsAdminException e){
               throw new JMSAdminException(e.getStackTrace());
      }
  }
```

Dintre del codi del connector, quan es gestioni l'excepció del servidor JMS concret (en l'exemple *TibjmsAddminException*), s'haurà de escalar l'excepció a través d'una nova *JMSAdminException*. Això vol dir, que en el codi del client monitor, s'aixecarà aquesta excepció i no s'haurà de gestionar directament l'excepció del servidor JMS; no es pot fer un codi dependent del servidor JMS amb que s'està treballant. Les excepcions seran mostrades a l'usuari com una finestra d'error perquè sigui informat del problema.

CAPÍTOL 6 Implementació

6.1 ESTRUCTURA DE FITXERS I DIRECTORIS

La implementació del monitor té dos parts diferenciades; d'una banda la llibreria *JMSMonitorInterface*, que permet obtenir les dades dels diferents proveïdors suportats, i el propi client gràfic *JMSMonitor*, que utilitzarà la llibreria anterior per poder mostrar la informació a l'usuari i oferir el total de les seves funcionalitats.

6.1.1 JMSMonitorInterface

Consta de tres directoris principals; *bin*, *lib* i *src*. Dintre de *src* es troba el codi font de la llibreria, que està distribuït en diferents subdirectoris en funció dels *packages* Java creats. En aquest directori es troben dos fitxers que serveixen per llençar la compilació de la llibreria i muntar la llibreria en un fitxer *jar*[24].

Dintre de la carpeta *lib*, es troben les llibreries necessàries per poder compilar el codi font i a *bin* es trobaran les classes compilades.

Per executar el *script* que compila la interfície s'ha d'establir la variable d'entorn JAVA_HOME apuntant al directori principal de la instal·lació de la JDK, ja que es buscarà el compilador de Java "javac":

```
...
if [ -z "$JAVA_HOME" ] ; then
  echo "fatal error -> JAVA_HOME is not setted!"
...
```

Amb les llibreries incloses dintre de la entrega, es genera el *classpath* per poder dir al compilador de Java, quines són les llibreries a incloure per tal de compilar el codi:

```
    export OPENJMSLIB="$JMSINTERFACEHOME/lib/openjms"
    export ACTIVEMQLIB="$JMSINTERFACEHOME/lib/activemq"
    export CLASSPATH="$OPENJMSLIB/openjms-0.7.6.1.jar:$OPENJMSLIB/jms-1.0.2a.jar :
$OPENJMSLIB/jndi-1.2.1.jar"
    export CLASSPATH="$ACTIVEMQLIB/activemq-core-5.3.0.5-fuse.jar:..
```

Cal dir que amb tota ampliació d'aquesta part del projecte s'hauran de revisar els fitxers que automatitzen la compilació, afegint les noves llibreries utilitzades o nous paquets de Java desenvolupats per ser compilats.

24 ***Jar***: Un *java archive* és un format de arxiu utilitzat per empaquetar tots els components d'una aplicació o projecte Java.

Una vegada s'inicia la compilació, els fitxers *.class* són emmagatzemats al directori bin, per a que tot seguit es generi el fitxer *jar* per poder utilitzar-lo en la part gràfica:

```
export SRC_HOME="./org/jmsmonitor/bridge"
javac -cp $CLASSPATH -d ../bin $SRC_HOME/model/*.java $SRC_HOME/exceptions/*.java
$SRC_HOME/plugins/*.java $SRC_HOME/main/*.java
cd ../bin
jar cf ../JMSMonitorInterface.jar .
```

A més del *script bash*, també s'inclou un fitxer bat per poder compilar en entorns Windows.

6.1.2 JMSMonitor

La entrega de JMSMonitor consta de sis directoris; *bin*, *conf*, *icons*, *lib*, *sessions* i *src*. Els directoris *bin*, *lib* i *src* tindran el mateix paper que en la interfície. El directori *sessions* tindrà els fitxers amb les sessions de connexió del monitor emmagatzemades mentre que el directori *icons* contindrà les icones i imatges emprades pel client gràfic. Per tal de configurar les traces de l'aplicació, dintre de *conf* es trobarà el fitxer *log4j.properties*. Cal recordar que, qualsevol canvi en la llibreria *JMSMonitorInterface* no implica recompilar el codi de la part gràfica, únicament s'haurà d'incloure dintre del directori *lib*.

Al igual que amb la llibreria, s'inclouen *scripts* de compilació (tant per Linux com Windows) que necessitaran que la variable d'entorn JAVA_HOME estigui correctament declarada. Aquests són semblants als anteriorment descrits però cal destacat algunes diferencies a l'hora de generar el fitxer jar que servirà per poder executar l'aplicació.

Com s'observa a continuació, s'inclou un manifest dintre del jar que es generà a partir de les classes compilades:

```
jar cfvm $JMSMONITOR_HOME/lib/JMSMonitor.jar Manifest.txt
```

Aquest indicarà quina és la classe principal i quin és el *classpath* per tal de que es trobin totes les classes que utilitza JMSMonitor:

```
//Contingut del manifest
Class-Path: geronimo-jms_1.1_spec-1.1.1.jar jcommon-1.0.16.jar
JMSMonitorInterface.jar jfreechart-1.0.13.jar log4j-1.2.15.jar

Main-Class: org.jmsmonitor.JMSMonitor
```

El fitxer resultant del *script* de compilació es trobarà al directori *lib*. Per tant, l'usuari per executar l'aplicació haurà de situar-se al directori *bin* i executar el *script* que llença el monitor. Aquest requereix dos paràmetres per indicar l'idioma de la aplicació:

```
jordi@jordi-laptop:~/Escriptori/JMSMonitor/bin$ JMSMonitor.sh en us
Checking environment...
JMS MONITOR HOME: /home/jordi/Escriptori/JMSMonitor
java -jar /home/jordi/Escriptori/JMSMonitor/lib/JMSMonitor.jar en us
2010-08-22 12:47:03,159 [INFO ] org.jmsmonitor.JMSMonitor > Starting JMSMonitor...
2010-08-22 12:47:03,161 [DEBUG] org.jmsmonitor.JMSMonitor > JMSMONITOR_HOME =
/home/jordi/Escriptori/JMSMonitor
2010-08-22 12:47:03,561 [DEBUG] org.jmsmonitor.gui.frames.MainFrame > Starting main window...
2010-08-22 12:47:03,561 [DEBUG] org.jmsmonitor.gui.frames.MainFrame > setting look&feel...
2010-08-22 12:47:03,996 [DEBUG] org.jmsmonitor.gui.frames.MainFrame > setting locale...
2010-08-22 12:47:04,003 [DEBUG] org.jmsmonitor.gui.frames.MainFrame > creating interface components...
2010-08-22 12:47:04,078 [DEBUG] org.jmsmonitor.gui.elements.MainMenu > disabling charts...
...
```

Tal com s'inicia l'aplicació les traces començaran a donar informació sobre els processos interns de l'aplicació i possibles advertències i errors que es vagin donant, com es mostra en l'exemple anterior.

6.2 CONFIGURACIÓ DE LA APLICACIÓ

En aquest apartat es resumeixen totes aquelles decisions preses sobre els aspectes relatius a la configuració de l'aplicació i com l'usuari podrà modificar certs paràmetres per tal de personalitzar l'eina.

6.2.1 Traces de l'aplicació

Tota aplicació necessita mostrar o registrar informació dels processos interns a la mateixa i de cadascuna de les tasques que realitza durant el seu funcionament per facilitar treballs de depuració i detecció d'errors de l'aplicació. Per realitzar una gestió intel·ligent i fàcilment configurable s'opta per l'ús de la llibreria *log4j*, un dels nombrosos projectes de la *Apache Software Foundation*. representant un cost econòmic mínim al projecte per la seva llicència *Apache License*.

Log4j anirà imprimint per pantalla, escrivint en fitxer o inclús fent insercions a bases de dades, missatges que es definiran en el codi en punts del flux d'execució que es considerin importants o crítics. Per tal de no inundar els registres amb missatges de traces o millorar el rendiment de l'aplicació, *Log4j* permet establir diferents nivells de seguretat (de menor a major detall):

- FATAL: s'utilitza per a missatges crítics del sistema, generalment després de guardar el missatge el programa fallarà.

- ERROR: s'utilitza en missatges d'error de l'aplicació que es vol guardar, aquests esdeveniments afecten el programa però el deixen continuar funcionant, com per exemple que algun paràmetre de configuració no és correcte i es carrega el paràmetre per defecte.

- WARNER: s'utilitza per a missatges d'alerta sobre esdeveniments que es vol mantenir constància, però que no afecten el correcte funcionament del programa.

- INFO: s'utilitza per a missatges similars a la manera "*verbose*" en altres aplicacions.

- DEBUG: s'utilitza per escriure missatges de depuració. Aquest nivell no ha d'estar activat quan l'aplicació es trobi en producció.

- TRACE: s'utilitza per mostrar missatges amb un major nivell de detall que *debug*.

La configuració de *Log4j* s'haurà de definir en un fitxer per indicar-li quin format tindran aquestes traces, la mida màxima del registre i una gran varietat de paràmetres que es poden consultar en la documentació oficial del projecte. A continuació es mostra un exemple del fitxer de configuració de log4j, on s'estableixen les propietats per les traces de DEBUG responent al següent format; *[5 espais per la prioritat del missatge] {categoria del missatge} data -> missatge de l'aplicació + retorn.*

```
  ########################################################################
   ### LOG4j CONFIG                                                     ###
   ########################################################################
log4j.rootCategory=DEBUG, CONSOLE
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=[%-5p] {%c} %x -> %m%n
```

En el codi de l'aplicació s'utilitzarà de la següent forma:

```
// obtenció de l'objecte logger
private static Logger log =
                  Logger.getLogger(ManageSessionForm.class.getName());
...
// mostrar missatge al log
log.debug("Load files...");
```

```
log.debug("Checking files...");
log.info ("Loading session OK");
```

Per tant en aquest cas, si el nivell en la configuració és DEBUG, el *log* mostrarà tots els missatges i si és INFO, només mostrarà l'últim. Cal afegir, que en el fitxer de configuració de *log4j* es pot redirigir la informació de log a fitxers i fins i tot a una base de dades, de tal forma que sempre es pot modificar sense recompilar codi el comportament global del logging de l'aplicació.

6.2.2 Multi idioma

Un dels altres aspectes importants que es té en compte és la possibilitat de canviar l'idioma dels diàlegs i menús de l'aplicació.

Per dur a terme aquesta característica de l'eina, s'utilitzarà la classe *ResourceBundle* de Java. Aquesta facilita la creació d'aplicacions traduïdes en diferents idiomes a més de poder utilitzar diferents escenaris al mateix temps. Per treballar amb aquesta classe caldrà definir fitxers de propietats amb l'extensió "*.properties*". Aquest fitxers contindran el següent patró:

```
# Messages_cat_ES.properties
camp1 = valor1
camp2 = valor2
camp3 = valor3
```

On ' # ' és un caràcter per indicar que es tracta de un comentari i '*valorN'* serà el contingut assignat a la variable '*campN'*. Tot seguit es mostra un exemple de la utilització de *ResourceBundle*:

```
//obtenir missatges en català
```

```
Locale l = new Locale("cat","ES");
ResourceBundle messages = (ResourceBundle.getBundle("Messages",l));
String text  = messages.getString("camp1");
```

El mètode estàtic *getBundle* retorna un *ResourceBundle* estàtic amb el contingut del fitxer *Messages_cat_ES.properties.* Una vegada ja s'ha carregat el contingut del fitxer en memòria, es poden obtenir els diferents valors que s'han assignat als camps de l'arxiu mitjançant el mètode *getString,* que com a paràmetre d'entrada espera el nom del camp del que es vol obtenir el valor.

Per aquesta fase del projecte, s'alliberen tres fitxers *Messages_cat_ES.properties*, *Messages_esp_ES.properties*, *Messages_en_US.properties* on es definiran els missatges i cadenes de text en català, castellà i anglès respectivament.

6.2.3 Icones i imatges

Per les icones i imatges emprades en la interfície gràfica s'ha optat per *Tango*, una llibreria gratuïta i oberta d'icones força utilitzada en entorns d'escriptori Linux com Gnome o KDE. A partir de la versió 0.8.90 és de domini públic mentre que les antigues tenen un llicència *Attribution-ShareAlike 2.5 Generic* de *Creative Commons*.

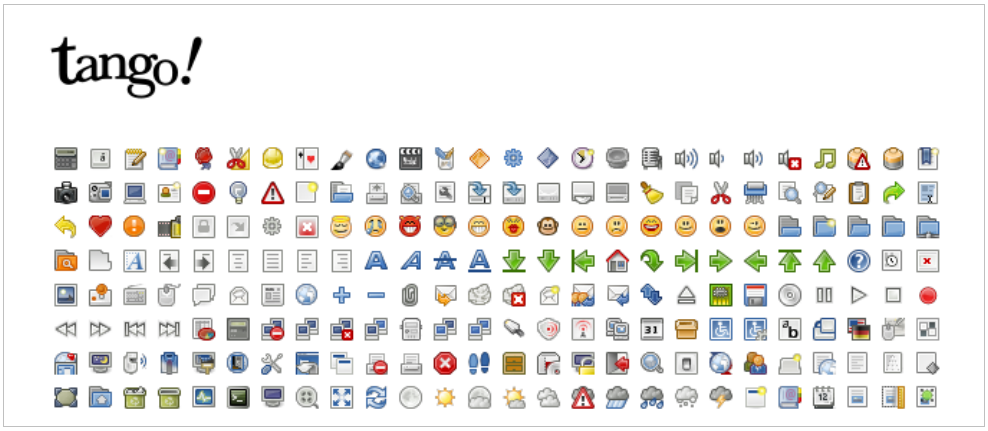


Figura 37: llibreria d'icones Tango

Únicament s'inclouen a la versió alliberada del present projecte aquelles icones que s'han utilitzat i no tota la llibreria sencera. S'empren dos conjunts de mides, una de 16x16 píxels, molt apropiada per menús i capçaleres de finestres, i l'altre de 22x22 més orientada a botons.

Swing permet afegir icones als seus elements de forma fàcil a través dels mètodes de la classe en qüestió. Per exemple, a la classe *MainMenu* dintre de la package *org.jmsmonitor.gui.elements* es declaren les entrades que tindrà el menú principal de la aplicació. Aquests tindran una petita icona per identificar cada opció amb una acció concreta de forma més visual:

```
   ...
   //create entries of the "ACTIONS" component
String path = this.homepath + fsep + "icons" + fsep + "small" + fsep + "_purge.png";
ImageIcon iconPurge = new ImageIcon(path);
JMenuItem actionsPurge = new JMenuItem(messages.getString("MenuActionsPurge"),
                                       iconPurge);
actionsPurge.setToolTipText(messages.getString("MenuActionsPurgeHelp"));
   ...
```

Cal anotar que en el codi anterior es fa servir la variable d'entorn JMSMONITOR_HOME i una cadena on es guarda el valor del separador que utilitza aquell sistema operatiu per construir les rutes (per exemple, a sistemes *Linux* s'utilitza el caràcter '/'):

```
private String homepath = System.getenv("JMSMONITOR_HOME");
private String fsep = System.getProperty("file.separator");
```

6.3 INTERFÍCIE D'ADMINISTRACIÓ

Com s'ha comentat al apartat de disseny, separar el codi del client gràfic del monitor de la gestió de la connexió als diferents servidors és una part fonamental del projecte. Per tal de mantenir la màxima independència en el codi del client, de les classes que adapten les diferents interfícies dels múltiples servidors JMS, s'ha optat pel patró de disseny bridge.

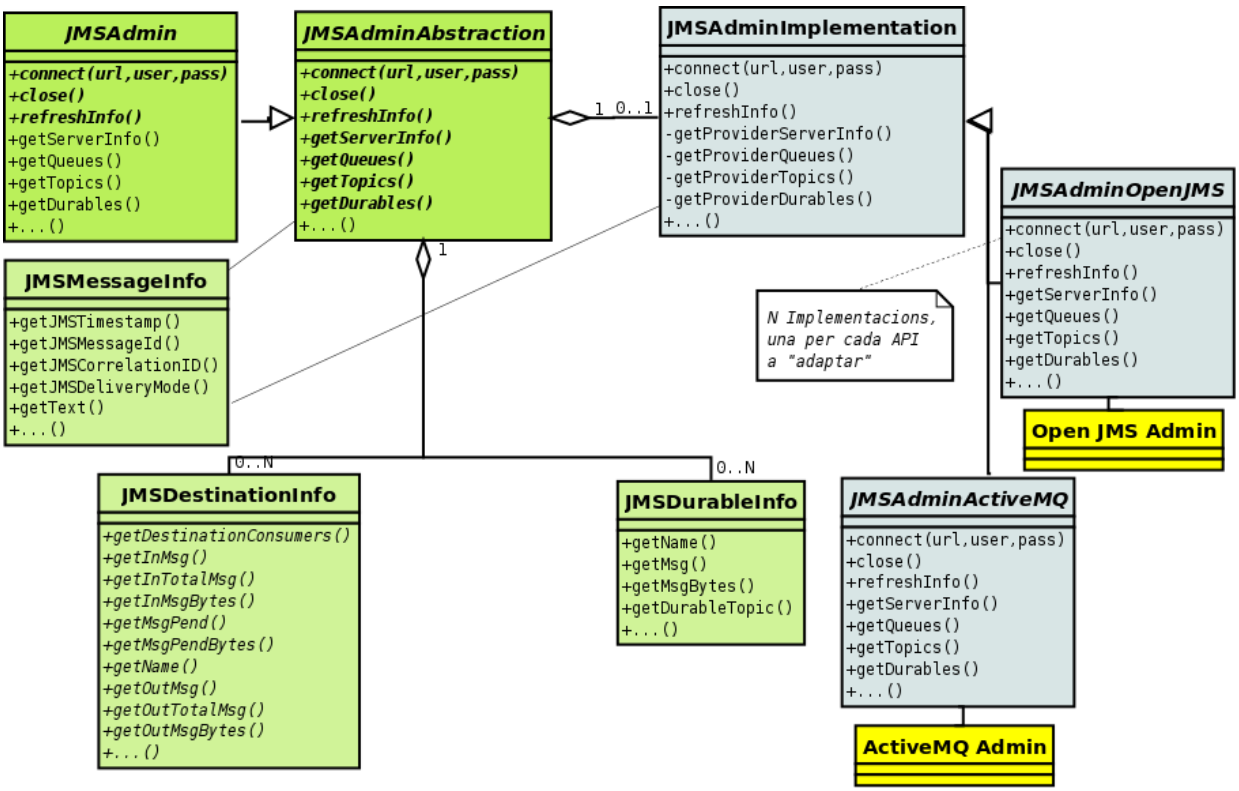


Figura 38: diagrama de classes amb el patró de disseny Bridge

La interfície està integrada per 4 *packages* que contenen les classes mostrades a l'anterior diagrama. A continuació es mostrà el nom d'aquests amb els components que tenen inclosos:

PACKAGE	CLASSES / INTERFACES
org.jmsmonitor.bridge.main	JMSAdmin JMSAdminAbstraction (abstract) JMSAdminImplementor (interface)
org.jmsmonitor.bridge.model	JMSDestinationInfo JMSDurableInfo JMSMessageInfo
org.jmsmonitor.bridge.plugins	JMSAdminActiveMQ JMSAdminArjunaMQ JMSAdminBossMessaging JMSAdminFioranoMQ JMSAdminHornetQ JMSAdminJBossMQ JMSAdminJORAM JMSAdminOpenJMS JMSAdminSonicMQ JMSAdminTIBCOEMS JMSAdminWeblogicMQ JMSAdminWebsphereMQ
org.jmsmonitor.bridge.exceptions	JMSAdminException JMSAdminConnectionException JMSAdminNonAvailableMethod

Taula 17: estructura de JMSMonitorInterface

org.jmsmonitor.bridge.main

JMSAdminAbstraction és una classe abstracta on es descriuen i defineixen els mètodes genèrics que utilitzarà el client (en aquest cas, la interfície gràfica del monitor) per accedir a les funcionalitats de l'API d'administració del servidor JMS. Inclou la interfície *JMSAdminImplementor* i les classes *JMSDestinationInfo*, *JMSDurableInfo i JMSMessageInfo*. A més, es defineixen unes constants per identificar els diferents proveïdors de JMS que existeixen en el mercat. D'entrada s'han escollit un subconjunt de 12 proveïdors bastant comuns en els desenvolupaments amb

tecnologia JMS:

```
//Supported providers
public final static int OPENJMS        = 1;
public final static int ACTIVEMQ       = 2;
public final static int TIBCOEMS       = 3;
public final static int HORNETQ        = 4;
public final static int FIORANOMQ      = 5;
public final static int JBOSSMQ        = 6;
public final static int JBOSSMESSAGING = 7;
public final static int JORAM          = 8;
public final static int ARJUNAMQ       = 9;
public final static int SONICMQ        = 10;
public final static int WEBLOGICMQ     = 11;
public final static int WEBSPHEREMQ    = 12;
```

Aquesta és una subclasse d' *Observable*. Java proporciona un sistema basat en el patró de disseny *Observer* força senzill d'usar i molt útil quan s'ha de notificar un esdeveniment a diversos objectes. Així doncs, cada vegada que es cridi al mètode *refreshModel,* que s'encarrega d'actualitzar el model de dades d'un objecte de la classe JMSAdmin, es notificarà als observadors per a que puguin tornar a obtenir el model de dades d'aquest i així poder gestionar la informació en la part gràfica del sistema.

```
public abstract class JMSAdminAbstraction extends Observable{
...
}
```

JMSAdmin és una classe filla de *JMSAdminAbstraction*, que implementa els mètodes descrits en la seva classe pare i per tant, les seves instàncies seran els objectes que crearà i utilitzarà el client del monitor per obtenir la informació. Així doncs, el client construirà l'objecte de la classe *JMSAdmin* indicant el tipus de proveïdor del que vol obtenir la informació:

```
JMSAdmin administrador = new JMSAdmin(JMSAdminAbstraction.ACTIVEMQ);
```

En el constructor, es crearà el nou implementador (*JMSAdminImplementor*) pel servidor en concret:

```
public JMSAdmin (int provider){
        switch(provider){
         case JMSAdminAbstraction.ACTIVEMQ:
                  implementor = new JMSAdminActiveMQ();
                  break;

         case JMSAdminAbstraction.ARJUNAMQ:
                  implementor = new JMSAdminArjunaMQ();
                  break;
        ...
```

Com s'ha comentat anteriorment, en la classe *JMSAdmin* es farà la notificació als observadors:

```
public void refreshModel() throws JMSAdminException{
 ...
  this.tDurables = durableInfo.length;
  setChanged();
  notifyObservers();
 ...
}
```

Degut a que la interfície tindrà només un objecte JMSAdmin i serà un recurs compartit per diversos objectes, s'haurà d'assegurar la mútua exclusió en algunes situacions que es tractaran més endavant, quan es parli del client gràfic.

JMSAdminImplementor és la interfície que connectarà les classes que utilitza el client amb aquelles que implementaran els mètodes necessaris per adaptar les API dels diferents proveïdors JMS.

org.jmsmonitor.bridge.model

JMSDestinationInfo, JMSDurableInfo i JMSMessageInfo són classes que representen la pròpia naturalesa de la informació que es tracta; informació sobre destinacions (cues / tòpics) , dades sobre els durables creats i el model de dades d'un missatge respectivament. Aquest últim s'utilitza per representar els camps del missatge que s'utilitzaran des del client gràfic sense emprar la interfície *Message* de la API de JMS; això permet desacoblar el codi del client d'aquestes llibreries i guanyar independència envers les diferents implementacions de JMS. És a dir, el client mai haurà de cridar a mètodes de la API JMS, si no que utilitzarà sempre la present interfície per obtenir les dades que necessiti.

org.jmsmonitor.bridge.plugins

En aquesta *package* s'inclouen totes les classes que serveixen per connectar el monitor amb les classes pròpies del proveïdor objectiu; transformen el model de dades que s'obté mitjançant les API d'administració de les diferents implementacions de JMS al model de dades amb que treballa el monitor. En aquesta versió no s'implementen totes, però per concordança amb les constants definides en la classe *JMSAdminAbstraction*, s'inclouen en el projecte sense la funcionalitat desenvolupada. Les implementacions disponibles són les de *ActiveMQ* i *OpenJMS* on més endavant es detallaran.

org.jmsmonitor.bridge.exceptions

S'inclouen totes les excepcions produïdes per les diferents implementacions de JMS. Aquestes seran gestionades pel client monitor, per tant, en el codi dels connectors o de les classes que formen la interfície mai es farà un tractament d'aquestes excepcions concretes

dependents de la implementació JMS, si no que es propagaran les genèriques que inclou aquest paquet per a que el client pugui determinar l'acció a realitzar (mostrar un missatge per pantalla, sortir de la connexió, etc.). En el següent exemple es mostrà com una excepció que llença el proveïdor de JMS s'escala a una excepció definida en la *package* d'excepcions de la interfície:

```
  public void connect(String URL, String USER, String PASSWORD)
                                throws JMSAdminConnectionException {

    try {
         admin = AdminConnectionFactory.create(URL, USER, PASSWORD);
     } catch (MalformedURLException e) {
          throw new JMSAdminConnectionException(e.getStackTrace());

     } catch (JMSException e) {
          throw new JMSAdminConnectionException(e.getStackTrace());
          }
}
```

Es pot trobar el cas, en que la implementació d'un connector sigui incompleta o que hi hagi algun mètode que no estigui desenvolupat. La classe *JMSAdminNonAvailableMethod* permetrà gestionar aquest tipus de situacions. En el codi del connector es trobarà el mètode que té una funcionalitat no suportada.

Així doncs en el codi de la interfície gràfica, es podrà tractar aquesta excepció tot indicant a l'usuari el per què del error:

```
...
try{
      log.debug("managing new connection...");
      GUI.getSession().getConnection().connect();
      ...
 } catch (JMSAdminNonAvailableMethod f){
   log.error(Utils.StackTrace2String(f.getStackTrace()));
   GUI.showError(...);
}
...
```

Connector per OpenJMS

Abans de començar la implementació, el servidor OpenJMS ha de configurar-se per tal de definir els paràmetres per la connexió; s'ha d'editar el fitxer *<openjmshome>/config/openjms.xml* que s'entrega per defecte (on *<openjmshome>* és el directori principal de la instal·lació del producte), ja que si no serà impossible connectar-se al servidor perquè no està definit per acceptar connexions TCP:

```
...

<Configuration>
  <!-- Optional. This represents the default configuration  -->
  <ServerConfiguration host="192.168.1.45" embeddedJNDI="true" />

  <Connectors>
  ...
 <Connector scheme="tcp">
     <ConnectionFactories>
       <QueueConnectionFactory name="TCPQueueConnectionFactory"/>
       <TopicConnectionFactory name="TCPTopicConnectionFactory"/>
     </ConnectionFactories>
   </Connector>
...
  </Connectors>
 <TcpConfiguration port="3030" jndiPort="3035"/>
<SecurityConfiguration securityEnabled="true"/>
 <Users>
   <User name="admin" password="admin"/>
  </Users>
...
```

Com es pot veure en l'anterior fitxer, es defineix la IP del servidor i el port (3030 en aquest cas) a més d'habilitar la seguretat del servidor indicant quins usuaris i amb quina clau podran accedir.

En la implementació de la classe *JMSAdminOpenJMS*, que és la que connecta els mètodes

del servidor amb la aplicació a desenvolupar, es farà servir una interfície que proporciona el proveïdor per tasques de administració:

```
admin = AdminConnectionFactory.create(URL, USER, PASSWORD);
```

S'ha pogut comprovar que es recomanable incloure el protocol al passar la direcció de connexió del servidor (p.e; ***tcp***://localhost:3030). Hi ha implementacions JMS on no es cal especificar i amb indicar només la direcció i el port és suficient, però a *OpenJMS* és necessari.

Respecte als mètodes d'enviament i neteja de missatges (consistent en un client que consumirà tots els missatges d'una cua), s'utilitza JNDI per establir la connexió amb la *factory*. En el següent fragment es poden veure les diferencies a l'hora de treballar amb l'API de OpenJMS entre tòpics i cues:

```
public void sendTextMessage...{
 context = loadInitialContext(initcontext, jndiServerUrl);
  ...
 if (!isTopic){
   QueueConnectionFactory qfactory = (QueueConnectionFactory)
                                          context.lookup(factory);
    ...
  }
 else{
   TopicConnectionFactory tfactory = (TopicConnectionFactory)
                                             context.lookup(factory);
    ...
  }
```

En el mètode *loadInitialContext* és retorna un objecte *Context* que serveix com a punt d'entrada al sistema de noms de Java. En el paràmetre *initContext*, que l'usuari haurà d'especificar mitjançant la interfície gràfica, s'espera el valor que el proveïdor JMS ens indica a la

documentació, per *OpenJms* és "*org.exolab.jms.jndi.InitialContextFactory*".

Per les dades de les cues i destinacions, OpenJMS no disposa d'un mètode explícit per obtenir cadascuna d'aquestes per separat, si no que s'han d'obtenir totes i després dividir-les en funció de la seva classe (veure mètode *splitDestinations* de la classe *JMSAdminOpenJMS*).

Durant la implementació dels mètodes que connecten amb el servidor de *OpenJMS*, s'han observat algunes mancances de la API d'aquest proveïdor JMS sobretot a nivell de estadístiques generals del servidor (número de consumidors, missatges entrants i missatges sortints). Aquest fet ha condicionat el desenvolupament del connector, ja que hi han dades estadístiques que no són calculades i no es podran utilitzar en el client gràfic del monitor (es retornaran els valors sempre a 0). No obstant permet realitzar algunes funcionalitats bàsiques; obtenir el nom de cues, tòpics, durables i crear usuaris i destinacions. Una possible solució seria revisar el codi de *OpenJMS* ja que és una aplicació de codi obert, però està fora del abast del present projecte. En concret els mètodes i dades que no es poden obtenir pel seu ús en el monitor són:

- Missatges d'entrada i de sortida en el servidor: no hi ha cap mètode per obtenir-los. Això també implica al ràtio calculat en cada refresc del nous missatges entrants i sortints.

- Nombre de consumidors

Aquesta limitació de la API de *OpenJMS* afecta a la funcionalitat de l'aplicació i es podria revisar en un futur lliurament o evolució de la mateixa.

Connector per ActiveMQ

Per tal d'obtenir els paràmetres a monitoritzar de ActiveMQ s'ha utilitzat JMX. *Java Manage eXtensions* és una tecnologia que defineix una arquitectura de gestió, API, patrons de disseny i

serveis per la monitorització de aplicacions desenvolupades sota Java. Aquest fet, no va ser contemplat al principi del projecte i es van realitzar les modificacions pertinents per incloure aquest nou requeriment. Cal dir que altres proveïdors JMS també utilitzen aquesta tecnologia i per tant no és un recurs especific per un connector, si no que pot ser extensible en el desenvolupament d'altres. S'afegeix doncs a la interfície de connexió amb els proveïdors un mètode per dir si el connector utilitzarà JMX o no. D'aquesta manera, la interfície gràfica podrà distingir quines dades són les que necessita que l'usuari ompli en el gestor de connexions per tal de poder establir una sessió JMX contra el servidor JMS.

En el servidor es podrà configurar l'accés JMX amb autenticació o sense. Si no es requereix aquest nivell de seguretat bastaria amb comprovar que al fitxer de configuració de *ActiveMQ* apareix una configuració equivalent a la següent:

```
...
 <broker xmlns="http://activemq.org/config/1.0" useJmx="true"
    brokerName="localhost"
    dataDirectory="${activemq.base}/data">
    <managementContext>
       <managementContext connectorPort="2011" jmxDomainName="my-broker"/>
    </managementContext>
    <!-- The transport connectors ActiveMQ will listen to -->
    <transportConnectors>
...
```

En canvi si es vol tenir restringit l'accés s'hauran de realitzar les següents accions:

· Canviar el fitxer anterior amb aquest valor:

```
<managementContext>
   <managementContext createConnector="false"/>
</managementContext>
```

· Crear dos fitxers; *jmx.access* i *jmx.password* dintre de la carpeta de configuració de ActiveMQ. Aquest últim haurà de tenir permisos 600 (escriptura i lectura només pel propietari) per

a que el servidor *ActiveMQ* arranqui correctament:

conf/jmx.access:

```
# The "monitorRole" role has readonly access.
# The "controlRole" role has readwrite access.
monitorRole readonly
controlRole readwrite
```

conf/jmx.password:

```
# The "monitorRole" role has password "abc123".
# The "controlRole" role has password "abcd1234".
monitorRole abc123
controlRole abcd1234
```

· Canviar el valor de la variable *SUNJMX* (dintre del fitxer */bin/activemq*) per incloure els dos fitxers anteriors:

```
  if [ -z "$SUNJMX" ] ; then
  #SUNJMX="-Dcom.sun.management.jmxremote.port=1099
-Dcom.sun.management.jmxremote.authenticate=false
-Dcom.sun.management.jmxremote.ssl=false"
  #SUNJMX="-Dcom.sun.management.jmxremote"
  SUNJMX="-Dcom.sun.management.jmxremote.port=1616
-Dcom.sun.management.jmxremote.authenticate=true
-Dcom.sun.management.jmxremote.ssl=false \
  -Dcom.sun.management.jmxremote.password.file=${ACTIVEMQ_BASE}/conf/jmx.password
\
  -Dcom.sun.management.jmxremote.access.file=${ACTIVEMQ_BASE}/conf/jmx.access"
  fi
```

Com a ajuda per treballar amb JMX també s'ha utilitzat *jconsole,* una utilitat proporcionada per Java, que permet navegar pels diferents valors i paràmetres de les aplicacions que poden treballar amb JMX.

Així doncs, per establir la sessió d'administració als servidor ActiveMQ, es carregaran les

credencials proporcionades per l'usuari i mitjançant la direcció i el nom de l'objecte JMX s'obtindrà una instància de *BrokerViewBean*, una classe que permetrà administrar el servidor. Mitjançant la classe *MbeanServerConnection* (connection) es podran obtenir les dades sobre destinacions. Tot seguit s'inclou un exemple per establir una connexió del plugin *d'ActiveMQ*:

```
  JMXServiceURL urlJMX;
  try {
        ...
       urlJMX = new JMXServiceURL("service:jmx:rmi:///jndi/"+ url +"/jmxrmi");
       connector = JMXConnectorFactory.connect(urlJMX, credentials);
       connector.connect();
       connection = connector.getMBeanServerConnection();
       ObjectName name = new ObjectName(this.objectName);

       adminBean = (BrokerViewMBean)
                            MbeanServerInvocationHandler.newProxyInstance(
                            connection, name, BrokerViewMBean.class, true);
      ...
}
```

El valor de la cadena *objectName* tindrà forma que s'indica a continuació, on "*localhost*" serà substituït pel nom (*BrokerName*) que se li ha donat en la configuració a l'intermediari que ens proporcionarà les dades del servidor (es pot veure anteriorment a aquest capítol quan es parlava sobre la configuració *d'ActiveMQ*):

```
"org.apache.activemq:localhost,Type=Broker"
```

Cal notar que la declaració d'aquest mètode és diferent a la de OpenJMS que té un tipus de paràmetres diferent, ja que aquest només necessita el nom de l'usuari, la direcció i la contrasenya (dades bàsiques per a una connexió administrativa). El connector de *ActiveMQ* només suporta l'administració mitjançant JMX, per tant, si des de el client s'invoqués a aquest mètode s'ha

d'informar que no està disponible:

```
   public void connect(String url, String user, String pass)
                                throws JMSAdminConnectionException,
JMSAdminNonAvailableMethod {
      throw new JMSAdminNonAvailableMethod("Only JMX connection is available");

}
```

Amb la API d'administració de *ActiveMQ* si s'han pogut obtenir totes les dades que necessita el monitor per complir amb les especificacions dels tòpics, cues i durables:

```
        ...
        ObjectName [] queueNames = adminBean.getQueues();
        this.queues = new JMSDestinationInfo[queueNames.length];

        for (int i=0;i<queueNames.length;i++) {
             QueueViewMBean queueMbean = (QueueViewMBean)
                     MBeanServerInvocationHandler.newProxyInstance(connection,
                                  queueNames[i], QueueViewMBean.class, true);
            this.queues[i] = new JMSDestinationInfo();
            this.queues[i].setName(queueMbean.getName());
            ...
```

En el l'exemple anterior, mitjançant l'objecte *adminBean* s'obtenen el nom de les cues en un vector. Per obtenir les dades relatives a cada cua es recorrerà el vector obtenint una instància de la classe *QueueViewMBean* que permetrà accedir a les dades de la cua corresponent a cada iteració. Per tòpics i per durables el mètode és semblant però utilitzant les classes *TopicViewMBean* i *DurableSubscriptionViewMBean* respectivament.

En quan a la creació i destrucció de destinacions, s'utilitza l'objecte *adminBean* per realitzar aquestes gestions. Per exemple, en la creació de tòpics i cues, s'utilitzaria el mètode *addTopic* i *addQueue* respectivament.

Per poder realitzar les funcions d'enviament de missatges i totes les funcionalitats que impliquen crear un client per tal de produir, consumir missatges o consultar les dades de les cues (*browse*), no s'utilitzarà JNDI fet que ens servirà per provar altra de les modificacions que es van realitzar al mateix temps que es creava el mètode per indicar si aquell connector utilitzava JMX o no.

Com que hi pot haver connectors que pot ser interessant utilitzar directament les llibreries pròpies del proveïdor JMS per connectar-se i enviar o rebre missatges, es crea un mètode estàtic a la classe *JMSAdminAbstraction* per indicar a la interfície gràfica si aquest connector requereix que l'usuari indiqui les dades JNDI a l'hora de configurar una sessió del monitor:

```
static public boolean isJMXPlugin(int p){...}
static public boolean usesJNDI(int p){...}
```

Aquestes funcions estàtiques (són relatives a la classe, no cal instanciar un objecte per utilitzar-les) truquen a altres mètodes estàtics que han de definir tots els proveïdors implementats. Per tant, s'afegeix al mòdul que fa d'adaptador entre el client gràfic i les API del diferents proveïdors, un mecanisme per donar certa flexibilitat a la interfície gràfica per decidir quina informació es requerida per treballar amb aquella connexió i quina altra no, i per tant, construir els formularis que l'usuari ha d'omplir per establir la connexió de forma adequada sense dades innecessàries.

Consideracions per a futures ampliacions

Per afegir nous proveïdors no definits en la classe JMS, en la package *org.jmsmonitor.bridge.plugins* s'hauran de crear les noves classes que implementin els mètodes per adaptar el codi dels administradors del proveïdors amb el model de dades del monitor JMS.

Conjuntament s'hauran d'editar i recompilar les classes:

- JMSAdminAbstraction: s'haurà de definir la nova constant que identifiqui al nou proveïdor a implementar, a més de la revisió dels mètodes estàtics comentats anteriorment per incloure els nous connectors dintre d'aquests.

- JMSAdmin: al constructor s'afegirà un nou cas en el selector per crear un implementador pel nou server a monitoritzar.

```
switch(provider){
        ...
      case JMSAdminAbstraction.NOUSERVERJMS:
            implementor = new JMSNOUSERVERJMS();
       break;

        ...
```

Com es veurà més endavant, el client gràfic és totalment independent i la inclusió o exclusió de connectors no afecten al codi d'aquest.

6.4 CLIENT GRÀFIC

Una vegada que ja s'ha desenvolupat la llibreria *JMSMonitorInterface*, és el torn de la implementació del client gràfic pel monitorizatge de servidors JMS. A continuació es nombren les diferents *packages*, classes i interfícies que conformen JMSMonitor:

PACKAGE	CLASSES / INTERFACES	
org.jmsmonitor	JMSMonitor (punt d'entrada)	
org.jmsmonitor.admin	Connection Session	Updater
org.jmsmonitor.gui.model	MainTableModel	ServerInfoModel
org.jmsmonitor.gui.elements	ButtonAutoRefresh ButtonConnection ButtonRefresh BrowseMessageFormEvents MainTableEvents	CommonList MainMenu MainServerInfo MainTabs MainToolbar PanelInfo
org.jmsmonitor.gui.frames	AbstractFrame (abstracte) AvailableSessionsForm BrowseMessageForm ClasspathForm ChartPieFrame Charts (interfície) ChartXYFrame Form	InfoFrame MainFrame ManageDestinationForm ManageSessionForm PurgeForm SendMSGForm
org.jmsmonitor.io	Printer SessionFile	SessionManager
org.jmsmonitor.utils	FileFilterCustom	Utils XMLManager

Taula 18: estructura de JMSMonitor

org.jmsmonitor

El punt d'entrada a l'aplicació es troba en aquesta *package* i espera dos paràmetres per indicar-li el idioma de la aplicació fent servir la classe *ResourceBundle* de Java, tal com s'ha comentat a l'inici d'aquest capítol. A més, es comprova que la variable d'entorn *JMSMONITOR_HOME* existeixi tot apuntant al directori actual del executable del monitor. Cal dir, que aquesta variable no cal que l'assigni l'usuari, si no que al *script* que llença l'aplicació ja s'estableix el valor correcte.

org.jmsmonitor.admin

Conté aquelles classes que s'utilitzen per establir una connexió amb els diferents proveïdors de JMS tot utilitzant la llibreria *JMSMonitorInterface*. Cal destacar la classe *Updater*, que és un *thread* que s'encarrega d'anar trucant al mètode de refresc de l'administrador *JMSAdmin* i per tant actualitzant el model de dades. Més endavant es descriurà aquest procés.

org.jmsmonitor.gui.model

Representa el model de dades del monitor. D'una banda, la informació general del servidor amb els totals acumulats com per exemple, el nombre total de missatges entrants i sortints, i per l'altra, les dades que conté la taula de la finestra principal del monitor.

org.jmsmonitor.gui.elements

Aquesta *package* recopila tots els elements que hi són dintre de les finestres del monitor

juntament amb les seves accions. Per exemple, aquí es trobaran la classe que representa el menú de la aplicació, que llença les diferents funcionalitats del monitor, o la barra d'eines que incorpora els botons pel refresc de les dades o per connectar/desconnectar-se del servidor.

org.jmsmonitor.gui.frames

Totes les finestres, inclosa la finestra principal de l'aplicació, es troba en aquest *package*. A més, també es recopilen els diferents formularis que l'usuari haurà d'omplir per determinades accions. Cal destacar també les classes que representen les gràfiques *ChartPieFrame* i *ChartXYFrame*, que implementen la interfície *Charts*. La primera seran les gràfiques de tipus circular i les segones seran gràfiques pintades sobre els eixos X i Y.

org.jmsmonitor.io

Les classes que es troben dintre d'aquest *package* s'encarreguen de totes les operacions de lectura i escriptura a disc. *SessionFile* representa un fitxer que emmagatzema una sessió mentre que *SessionManager* té els mètodes necessaris per salvar i carregar els fitxers que contenen sessions.

org.jmsmonitor.utils

Aquest paquet conté classes que implementes certes funcions i utilitzats de caràcter més general i no tan vinculat al monitor, com per exemple la verificació de XML a partir d'un XSD.

Una vegada s'ha descrit l'estructura general del *JMSMonitor*, a continuació s'aniran comentant els aspectes més rellevant de la implementació del client gràfic agrupant per funcionalitats generals o parts més diferenciades.

6.4.1 Funcionalitat bàsica

L'aplicació consta d'una **finestra principal** que tindrà un conjunt d'elements que podran disparar accions o representar dades. Aquesta representa la classe principal del client gràfic; controla i es comunica amb els seus components integrants. Quan es llença l'aplicació mitjançant el *script*, es crida a la classe que conté el mètode *main JMSMonitor*. Aquesta s'encarrega de comprovar els paràmetres de entrada, agafar la configuració de *log4j* i crear la finestra principal instanciant la classe *MainFrame*:

```
GUI = new MainFrame(Language,Country);
```

La classe *MainFrame* hereta de la classe abstracta *AbstractFrame*, que defineix mètodes que utilitzen totes les finestres com situar-les al centre de la pantalla, i altres més generals com la generació de missatges d'advertència, error i confirmació. *MainFrame* té els següents components:

```
private  MainMenu  miMenu;
private  MainToolbar miToolBar;
private  MainTabs  miTab;
private  List <Charts> misCharts;
private  MainServerInfo miInfoServidor;
private  Session session = null;
private ManageSessionForm sessionForm = null;
private ServerInfoModel serverInfoModel;
```

Com es pot observar, hi són el menú de l'aplicació, la barra d'eines, els tabuladors per escollir la taula a mostrar (cues, tòpics o subscripcions durables), una llista amb totes les gràfiques que són visualitzades en aquell moment i el panell amb la informació general del servidor juntament amb el seu model de dades. A la següent figura es mostra la representació gràfica d'aquests elements:

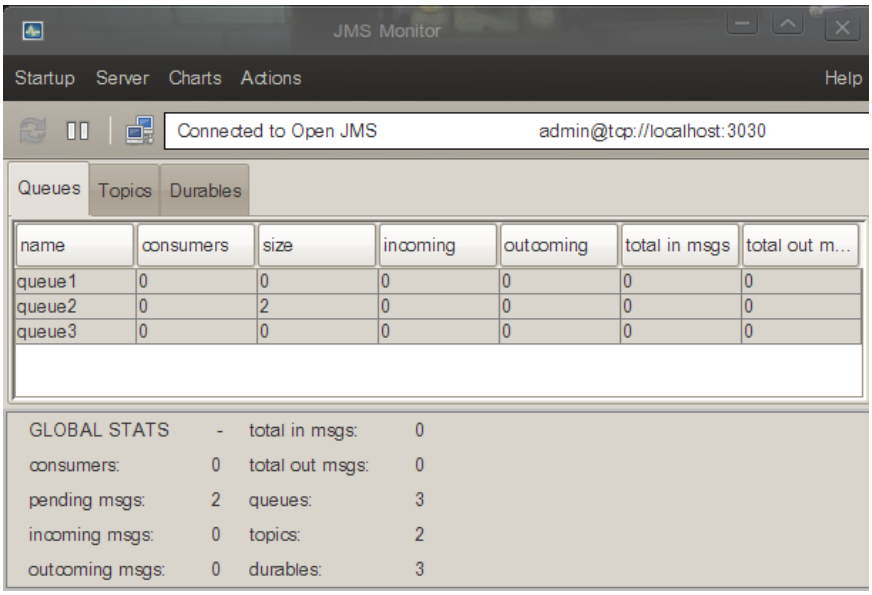


Figura 39: finestra principal de JMSMonitor

El menú principal de l'aplicació es representat per la classe *MainMenu*, que hereta de la classe *JMenubar* de la API de *Swing.* En el constructor d'aquesta classe es van afegint tots els menús i submenus que tindrà l'aplicació, podent-hi assignar icones i utilitzant els fitxers de text en funció de l'idioma en que s'hagi carregat l'aplicació. Amb el mètode *setToolTipText* s'afegeix un missatge per ajudar a l'usuari a entendre que farà aquella opció de menú; si passa el ratolí per sobre del menú apareixerà una descripció en d'ajuda de l'acció que s'executarà aquell ítem del menú. També es defineixen les accions que realitzaran aquest ítems quan l'usuari els seleccioni, de tal forma que es crearan noves finestres i formularis tal com es mostra en el següent exemple:

```
//action FILE -> EXIT
 fileClose.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent event) {
       GUI.exit();
   }
});
```

Cal dir, que els components del menú poden ser habilitats i deshabilitats en funció de quina sigui la situació del monitor. Per exemple, si no hi ha una sessió carregada, l'usuari no podrà seleccionar les opcions com “enviar missatge” o “crear cua” per evitar possibles errades.

La barra d'eines es representada per la classe *MainToolbar*, que hereta de la classe de la API de *Swing Jtoolbar*, conté tres botons i un camp per mostrar a quin servidor s'està intentant connectar:

```
 ...
 private ButtonConnection connection;
private ButtonRefresh refresh;
private ButtonAutoRefresh autoRefresh;
private JLabel lSessionHeader;
private JLabel lSessionInfo;
...
```

Al igual que passava amb el menú, aquests elements també poden ser habilitats i deshabilitats en funció de la situació del monitor.

Les dades de les cues, tòpics i durables representades per la classe *MainTabs* contindran també un menú contextual, el qual serà accessible amb el botó dret del ratolí. La classe *MainTableEvents* (hereta de la classe *MouseAdapter*) contindrà la implementació d'aquest menú.

```
  public void mouseClicked(MouseEvent arg0){
   ...
    int row = table.getSelectedRow();
    if (arg0.getClickCount() == 2 &&
        arg0.getButton() == MouseEvent.BUTTON1)
    {
        String name = (String) table.getValueAt(row,0);
        BrowseMessagesForm m = new BrowseMessagesForm(GUI);
        m.start(name,type);
    }
  ...
}
```

En fer *click* sobre la taula, s'invoca el mètode *mouseClicked*, que en funció de si s'ha realitzat amb el botó dret o és un doble amb el botó principal, mostrarà un menú o el contingut de la cua respectivament.

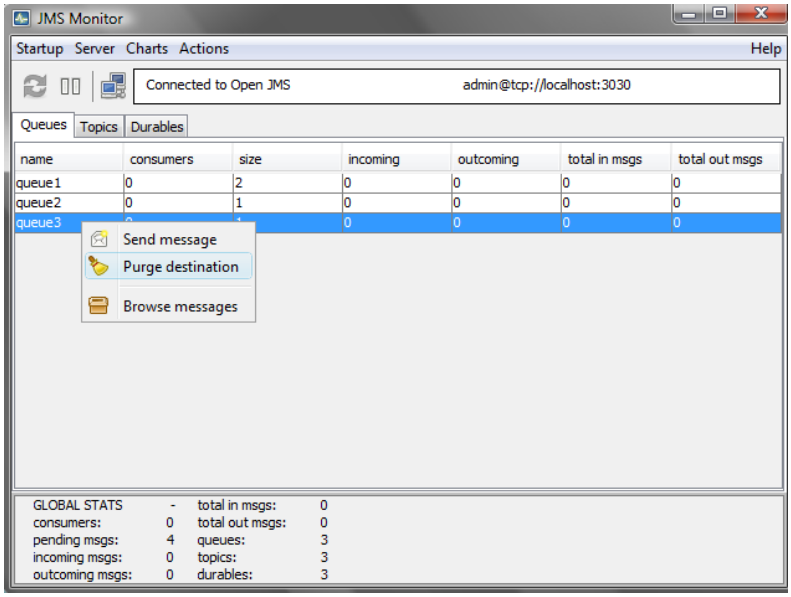


Figura 40: menú contextual de la taula principal

Cal dir, que en el constructor d'aquests elements, es passa com paràmetre la referència de la finestra principal de la aplicació *MainFrame*, per tal d'obtenir accés a altres objectes com per

exemple el mètode que connecta el monitor amb el servidor JMS (botó de connexió). Així doncs, el botó de refresc crida al mètode *refresh* de *MainFrame* (finestra principal) per tal d'obtenir una vegada les dades del servidor JMS i actualitzar el model de dades. En canvi, el botó d'habilitació del refresc automàtic llença un *thread* mitjançant la classe *Updater*, per tal de fer aquest refresc en la freqüència de milisegons que l'usuari ha definit en la sessió, mentre que el botó de connexió fa que s'inicii o es tanqui una connexió amb un servidor JMS. A continuació es detallaran aquestes funcionalitats.

Establiment de la connexió

Un dels components de la classe *MainFrame* és la classe *Session*, que representa les dades d'una sessió de connexió i gestiona junt amb els seus components, la comunicació amb el servidor JMS mitjançant la llibreria *JMSMonitorInterface*. La següent figura ajuda a entendre aquesta relació:

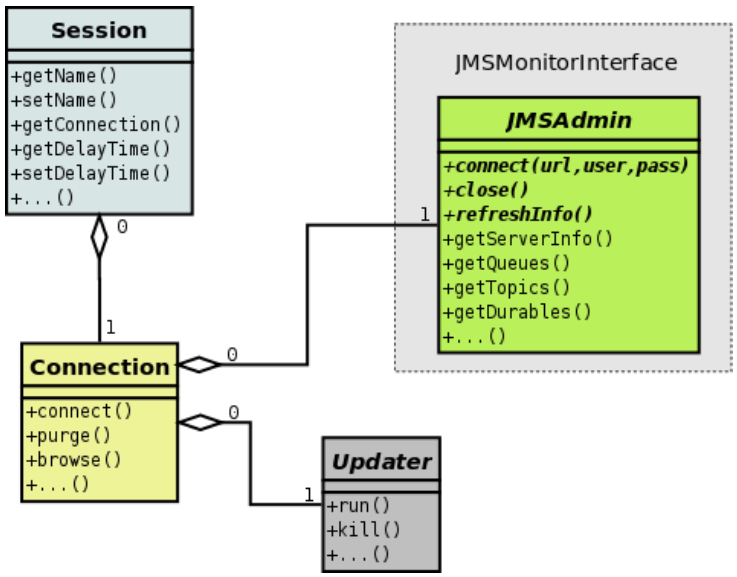


Figura 41: relació entre les classes que gestionen la connexió

El gestor de sessions (veure apartat "gestio de sessions") omple les dades l'objecte instanciat de Session juntament amb les dades de connexió. La classe *Connection* tindrà un component *JMSAdmin* (la classe del connector que s'ha implementat per tal de relacionar client i servidor JMS). Per tant, quan es vulgui iniciar una connexió es trucarà al següent mètode:

```
GUI.getSession().getConnection().connect();
```

En l'anterior exemple es pot veure com mitjançant la classe *MainFrame* es pot accedir a la sessió i a la connexió, per tal d'establir la connexió. El mètode *connect* utilitza les dades de la sessió per determinar quin tipus de connexió establir, si JMX o directament el mètode estàndard que proporcioni la implementació del connector:

```
public void connect() (...) {
 ...
 if (isJMX){
     admin.connect(host, user, pass, objectName);
   }
  else{
       admin.connect(host, user, pass);
  }
```

En cas que la connexió es perdi o el servidor no estigui disponible, es mostrarà una excepció a l'usuari ja que s'haurà aixecat l'excepció *JMSAdminConnectionException*, inclosa dintre del mòdul interfície *JMSMonitorInterface.*

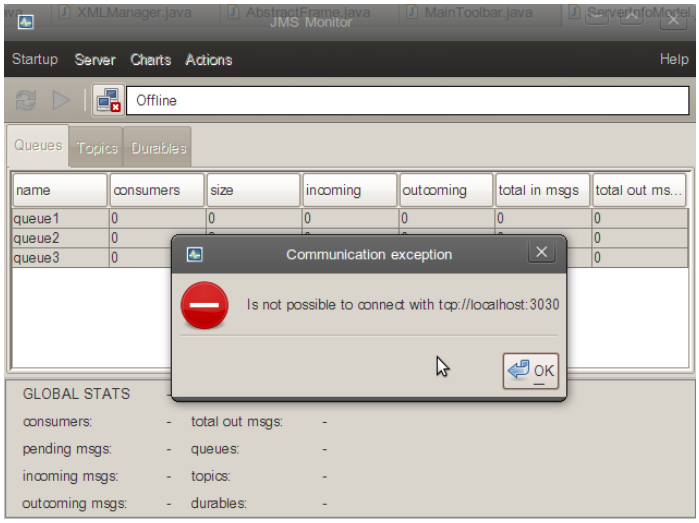


Figura 42: excepció de connexió

Refresc de les dades

En el mètode *connect* de la classe *Connection,* si la connexió s'estableix amb èxit s'afegeix a l'objecte *admin* (*JMSAdmin*) un observador. Aquest serà l'encarregat de dir a la resta de components que *JMSAdmin* ha actualitzat el seu model i que poden obtenir-lo per fer les seves operacions. Per poder dur a terme aquesta tasca s'ha de sobrescriure el mètode update en la classe *MainFrame* (implementa la interfície *Observer*). Aquest notificarà el canvi del model als diferents components com la taula principal, el propi botó de connexió (perquè canvii el seu estat), la informació general del servidor i les gràfiques per a que siguin pintades de nou amb els valors obtinguts.

Per l'actualització de la taula principal, s'invoca el mètode *refresh* de la classe *MainTabs*. Aquesta representa tres pestanyes on cadascuna conté una taula amb el model de dades monitoritzades per cues, tòpics i subscripcions durables. El model de dades es vincula amb la la vista en el constructor d'aquesta indicant el tipus de dades que contindrà mitjançant les constants de la mateixa classe *MainTableModel*:

```
public MainTabs(MainFrame GUI){
    this.GUI = GUI;
```

```
dtmQueues      = new MainTableModel(MainTableModel.QUEUE,
                                    GUI.getTextMessages());
dtmTopics      = new MainTableModel (MainTableModel.TOPIC,
                                    GUI.getTextMessages());
dtmDurables = new MainTableModel(MainTableModel.DURABLE,
                                    GUI.getTextMessages());
```

Per tal de recopilar les dades generals del servidor, es fa servir una classe per acumular aquestes anomenada *ServerInfoModel*. Quan *MainFrame* observa que hi ha una actualització en l'administrador, comptabilitza les dades de cada cua, tòpic i durable per tal de tenir un model amb el conjunt de dades global, que a més de ser representat en la vista principal del monitor, serà utilitzat per les gràfiques.

Refresc automàtic de les dades

D'altra banda, la classe *Updater* es llença cada vegada que s'estableix una nova connexió. Aquest no és més que un *thread* que truca al mètode que refresca el model de l'objecte *admin* (classe *JMSAdmin*).

Es important comentar el paper de *synchronized* en el moment d'invocar el mètode d'actualització del model de *JMSAdmin*, que és un recurs compartit i accessible a tots els elements que conformen el monitor. Així doncs, es pot donar el cas que en el moment que s'actualitza el model del administrador, l'usuari sol·liciti la creació/destrucció d'una cua. Això evidentment fa que el resultat del conjunt de les operacions sigui indeterminat; no es pot assegurar quin serà el nombre de cues comptabilitzar en el model, si l'anterior o el posterior a l'operació de creació/destrucció. El bloc *synchronized* s'executarà quan aquell objecte no estigui sent utilitzat per altre fil, assegurant així la mútua exclusió.

Amb el botó de connexió es destrueix el fil que actualitza les dades de forma automàtica i s'esborra la referència del component *updater.* En aquesta situació la interfície gràfica canviarà la seva vista per representar la desconnexió i no permetre que l'usuari fagi accions que si podia fer abans, acotant així possibles errors d'execució.

6.4.2 Gestió de les sessions

Per tal de poder carregar les dades de la connexió en el monitor, s'utilitza el gestor de sessions. Aquesta funcionalitat està representada per la classe *ManageSessionForm*, que visualitza un formulari (per omplir el model de dades *Session* i *Connection*) i quatre botons per tal de poder carregar, emmagatzemar, acceptar o cancel·lar una sessió. A més es poden incloure llibreries al *classpath* de forma dinàmica, per tal de poder connectar-se amb un servidor concret S'ha de recordar que cada proveïdor implementa les seves pròpies llibreries i que la interfície *JMSMonitorInterface* només assegura que el client "s'entengui" de forma transparent amb aquestes API. A continuació s'explicarà amb més detall les parts més rellevants del gestor de sessions.

emmagatzemar i recuperar sessions

Quan el botó per salvar la sessió es prem, s'activa el mètode per **emmagatzemar la sessió a disc**. El que farà es utilitzar la classe *SessionManager* per escriure a disc la informació del formulari.

```
   //SAVE
...
 sManager.loadSessions();
 sManager.saveSession(session);
 log.info("saved session " +
             session.getName() +" ok");
...
```

En l'anterior exemple, primer es carrega la informació general de les sessions que hi són ja guardades (mètode *loadSessions*), per veure si és una sessió nova o s'ha de reescriure. Aquesta informació està en un fitxer anomenat *sessions.monitor,* ubicat en el directori *sessions* de la

instal·lació del monitor. Té el següent format:

```
nom de la sessió = ruta del fitxer amb les dades de la sessió i connexió
```

Una vegada la classe *SessionManager* obté una llista amb la referencia dels noms de les sessions amb els fitxers, ja pot salvar la sessió al fitxer. Aquesta referencia ve representada per la classe *SessionFile*, que no és més que una subclasse de la classe *File* de Java, on s'inclou un camp amb el nom de la sessió.

Per salvar el fitxer, s'empra el mètode *saveConcreteSession* de *SessionManager* que utilitza la classe *Properties* de Java per muntant el fitxer amb el mètode *setProperty* com es mostra a continuació:

```
private void saveConcreteSession (...){
  Properties prop = new Properties ();
  OutputStream os = null;
  try{
      os = new FileOutputStream(file);
      Calendar cal = new GregorianCalendar();

      //CONNECTION
       Connection c = session.getConnection();
       prop.setProperty(FNAME,session.getName());
       prop.setProperty(FHOST, c.getHost());
       ...
       prop.store(os, (FCOMMENT + " SESSION"));
       ...
}
```

Quan l'objecte *prop* ha sigut establert assignant-li per cada etiqueta el seus valors corresponents, es crida al mètode *store* on se li passarà un objecte *OutputStream* per tal de crear el fitxer a disc. Cal dir que el nom que tindran les etiquetes dintre del fitxer per identificar cadascun

dels valors guardats de la sessió i connexió són constants definides en la classe:

```
   ...
   private static final String sFile = "sessions.monitor";
   private static final String FNAME = "session_name";
   private static final String FHOST = "connection_host";
   private static final String FUSER = "connection_user";
   private static final String FPASS = "connection_pass";
...
```

D'aquesta manera el fitxer de la sessió, anomenat com *nom_de_la_sessió.**monitor***, tindrà un contingut semblant al següent:

```
   ## SESSION
#Mon Aug 23 00:00:28 CEST 2010
connection_user=admin
jmx_objectname=
session_name=openjms_queue
isJMX=false
connection_initcontext=org.exolab.jms.jndi.InitialContextFactory
delay=2000
connection_jndiserver=tcp\://localhost\:3035
connection_pass=admin
cp=/opt/openjms_default/lib/jms.jar;/opt/openjms_default/lib/jndi
-1.2.1.jar;/opt/openjms_default/lib/openjms-0.7.6.1.jar
connection_provider=1
connection_factory=QueueCF
connection_host=tcp\://localhost\:3030
last=1282514362906
```

D'altra banda, qualsevol error que aparegui en el procés serà notificat a l'usuari aixecant una excepció en la vista del gestor de sessions (*ManageSessionForm*).

Per carregar una sessió, s'obrirà una nova finestra que mostrarà totes les sessions disponibles, que seran totes aquelles que estiguin registrades al fitxer *sessions.monitor*. La classe *AvailableSessionsForm* representa aquest formulari on l'usuari podrà carregar o esborrar una

sessió. En el constructor es passarà la referencia de la sessió que conté el formulari principal:

```
   public AvailableSessionsForm (MainFrame g, Session s){
     this.GUI = g;
     session = s;
     setTextMessages (GUI.getTextMessages());
     sManager = GUI.getSessionForm().getSManager();
}
```

Com es mostra a la següent figura, aquesta finestra conté un botó per esborrar, un altre per carregar la sessió al formulari del gestor de sessions i un altre per sortir.

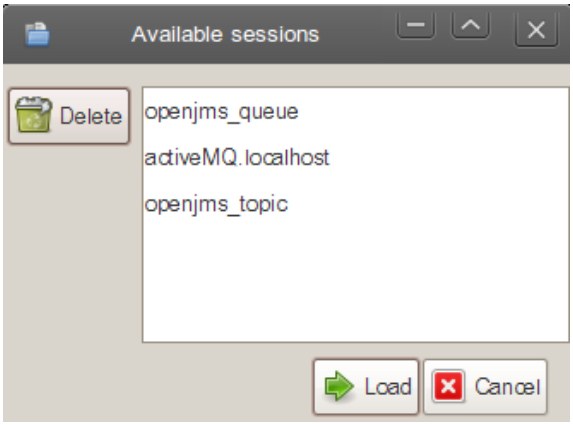


Figura 43: sessions guardades

En el cas de que **s'esborri una sessió emmagatzemada**, es cridara al mètode de la classe *SessionManager deleteSession* i es descartarà de la llista que es visualitza l'entrada corresponent:

```
   try {
      sManager.deleteSession(sf);
      listModelSessions.removeElement(sf.getSessionName());
   } catch (IOException e) {
     log.error("Session file not found: " +  sf.getPath());
    ...
}
```

Quan es vol carregar una sessió s'utilitza el mètode *getConcreteSession* de *SessionManager,* que retornarà un objecte *Session* amb les dades llegides del fitxer. Per tal d'actualitzar-lo al formulari principal del gestor de sessions, es copien les dades d'aquesta sessió obtinguda en la sessió referenciada en el formulari principal.

carregar la sessió en el monitor

El formulari principal conté una llista desplegable amb els proveïdors disponibles de *JMSMonitorInterface* (no necessàriament implementats). El nom d'aquest s'obté directament de la interfície fent més independent el client gràfic de la implementació dels connectors:

```
   int i = 1;
   String aux = JMSAdminAbstraction.getProvider(i);
   do{
       comboJMSServer.addItem(aux);
      i++;
      aux = JMSAdminAbstraction.getProvider(i);
  }while(aux.equals((Integer.toString
          (JMSAdminAbstraction.UNDEFINED))) != true);

// Determinar la activacion de los campos exclusivos para JMX con
el valor por default del JCOMBO
filterFieldsWithProviders();
...
```

En funció del proveïdor escollit, la vista del formulari canvia en funció de si aquell connector suporta *JNDI* o utilitza *JMX*, tal com es va comentar en la explicació de *JMSMonitorInterface*. El mètode *filterFieldsWithProviders* habilita o amaga els diferents camps del formulari en funció d'aquests paràmetres.

Una vegada que l'usuari ha introduït les dades o les ha obtingut des d'un fitxer, es poden carregar les dades al monitor, per tal de poder connectar-se al servidor. La primera acció que es realitzarà serà un comprovació de les dades que s'han introduït al formulari, per verificar que no hi

ha camps obligatoris que hi manquen o amb caràcters no vàlids. El mètode que fa aquesta comprovació en la classe *ManageSessionForm* és *parseSessionForm*.

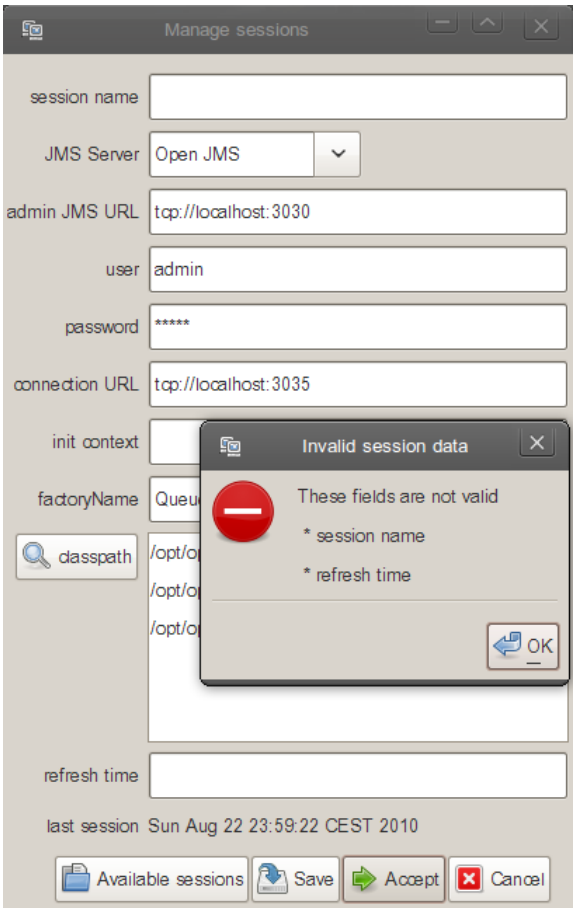


Figura 44: gestor de sessions

Com s'observa en la figura anterior, quan hi ha camps que no són vàlids, s'informa al usuari indicant el nombre dels camps. El mètode *ParseSessionForm* mira cadascun dels camps comprovant que aquests no estiguin buits i que no continguin caràcters que no estiguin dintre d'un rang determinat. Aquests mètodes es troben en la classe *Utils*, que recopila un conjunt de mètodes més globals:

```
// valid characters
if (!((aux >= 'a' && aux<='z') ||
      (aux >= 'A' && aux<='Z') ||
      (aux >= '0' && aux<='9') ||
      (aux == ' ') ||
```

```
            (aux == '_') ||
            (aux == '.'  ||
            (aux == ':' ))))
    return false;
```

Després de validar la informació, es comprova que no hi havia cap sessió prèvia. Si fos així es mostraria un missatge per advertit a l'usuari que es tancarà la sessió actual. Una vegada es realitzen totes les comprovacions, es carrega la informació del formulari en el monitor, creant un objecte *Session* amb les dades del formulari i assignant-lo a la classe *MainFrame* (pantalla principal del monitor):

```
    createSessionObjects();
    session.loadClasspath();
GUI.setSession(session);
```

Un altre punt a comentar és la càrrega de les llibreries adjuntades per l'usuari en el *classpath* de l'aplicació de forma dinàmica. Mitjançant la classe *URLClassLoader*, s'obtindrà la referencia a al carregador de classes del sistema per afegir les introduïdes per l'usuari:

```
URLClassLoader classLoader = (URLClassLoader)
                              ClassLoader.getSystemClassLoader();
```

En cas de que una llibreria ja estigui carregada, s'ignorarà per no fer-ho un segon cop. Cal dir que es va valorar la possibilitat d'implementar un carregador de classes múltiple, capaç de descarregar i carregar classes en temps d'execució, però no és quelcom senzill i s'escapa al volum d'hores de treball destinades al projecte. En una possible ampliació o revisió de la eina, podria ser un tema interessant a desenvolupar.

6.4.3 Funcionalitats amb destinacions

La aplicació afegeix funcionalitats addicionals per treballar amb destinacions i missatgeria. Aquestes són la consulta de missatges en cues, l'enviament de missatges a destinacions, crear i destruir destinacions i eliminar els missatges d'un destí. Tot seguit es detallaran els aspectes més important del desenvolupament d'aquestes.

enviament de missatges

La classe que s'encarrega de mostrar el formulari i realitzar la petició a l'administrador és *SendMSGForm*, que consistirà en una finestra amb un quadre de text per introduir el cos del missatge, una secció per seleccionar cua o tòpic, botons de cancel·lació, confirmació i associació de XSD.

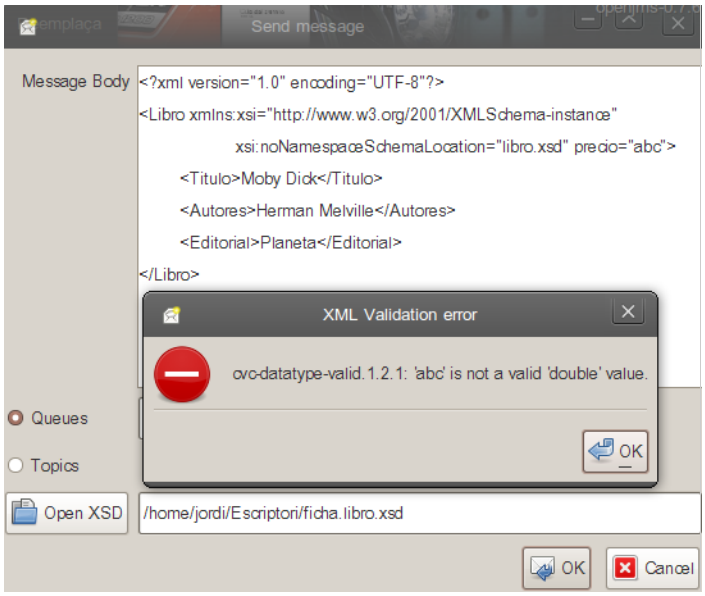


Figura 45: enviament de missatges i validació XML

Així doncs, s'utilitzarà el objecte administrador que conté la sessió actual per enviar un

missatge a una destinació, ja sigui cua o tòpic. Primer es comprovara si l'usuari a afegit un XSD a validar contra el cos del missatge; la classe *XMLManager* de la *package* del monitor *Utils*, s'encarrega de llençar una excepció en cas de que el XML no acompleixi amb l'especificació del XSD (veure imatge anterior):

```
...
String sXsd = "";
 XMLManager xmlManager = new XMLManager();
 sXsd = fxsd.getText();
 if (sXsd.length() > 0){
   //parseamos el XML
   log.debug ("parsing text with the XSD '" + fxsd.getText()
                                                    + "'");
   xmlManager.validate(fxsd.getText(),text.getText());
 }
```

Si la validació és correcta (o pel contrari no calia validar el contingut del missatge), es trucarà a l'administrador de la sessió per sol·licitar l'enviament del missatge. Amb una mateixa sentència es podrà diferenciar entre el que és un enviament a una cua i a un tòpic, a més de si usarà l'adreça JNDI o la de administració del proveïdors JMS; cal recordar que si el connector utilitza una de les dues adreces per l'enviament de missatges serà una dada acotada en el formulari per la informació que s'obté del propi connector:

```
c.getAdmin().sendTextMessage(
((c.getJNDIServer()==null||c.getJNDIServer().equals(""))?c.getHost():c.getJNDIServer()),
  c.getCurrentInitContext(), c.getCurrentFactory(), (String)comboDest.getSelectedItem(),
  text.getText(), c.getName(), c.getPass(), (rbuttonTopics.isSelected()));
```

Cal dir que el formulari d'enviament de missatges estarà disponible a través del menú contextual de la taula principal i el menú de l'aplicació. Si es selecciona aquest últim, el desplegable del formulari es situarà automàticament a la destinació que ja es trobava seleccionada en la pestanya corresponent visualitzada per l'usuari. Aquesta informació es

sol·licitarà a la classe *MainTab*, on els mètodes *getActiveTopicName*, *getActiveQueueName* i *getDurableName* retornaran el nom de la destinació seleccionada en cadascuna de les pestanyes.

neteja de missatges

Amb la classe *PurgeForm* es podran netejar els missatges continguts en cues i subscripcions durables. Amb una vista molt senzilla es presenta un selector per ambdós tipus de destinacions i una llista desplegable per seleccionar la cua o subscripció durable a buida, tal com es mostra a continuació:

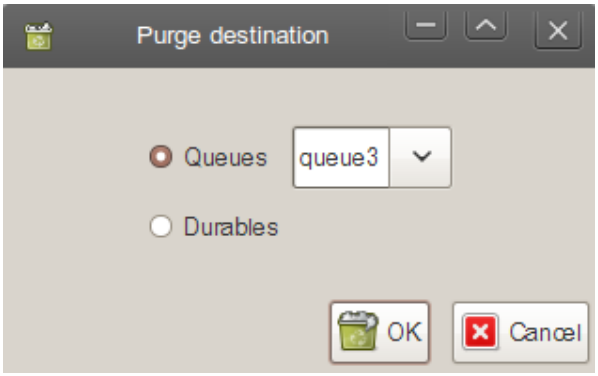


Figura 46: formulari de neteja de destinacions

En funció si la destinació te associada un tòpic o no es podrà determinar si és una cua o un tòpic; si té aquest últim es tractarà d'una subscripcions durable i es trucarà al mètode *purge* passant el nom d'aquest:

```
GUI.getSession().getConnection().purge(aux,null);
```

creació i eliminació de destinacions

Tant per crear i destruir destinacions, ja siguin cues, tòpic o durables s'utilitzarà una sola classe, *ManageDestinationForm*. En el constructor s'indicarà el tipus de destinació que és i l'acció disponible:

```
public ManageDestinationForm(MainFrame g,
                             int destination_type, int available_action)
```

Aquest dos últims enters, són constants definides en dues classes. El tipus de destinació es defineix a la classe que representa el model de la taula principal *MainTableModel* i l'acció disponible vindrà donada en la mateixa classe *ManageDestinationForm*:

```
public final static int CREATE = 0;
public final static int DESTROY= 1;
```

Per cada destinació doncs, es trucaran als mètodes de l'administrador de la sessió de forma diferent, utilitzant un selector pel tipus de destinació:

```
switch(destination_type){
    case (MainTableModel.QUEUE):
        if (action == CREATE)
            a.createDestination(dName.getText(), false);
        else
            a.destroyDestination(comboDest.getSelectedItem().toString(), false);
        break;

    case (MainTableModel.TOPIC):
        ...
```

Si al crear una destinació, el nom ja existeix, el monitor no deixarà que s'executi la trucada a l'administrador. Aquesta comprovació prèvia la realitza el mètode *parseName* del mateix formulari de gestió de destinacions.

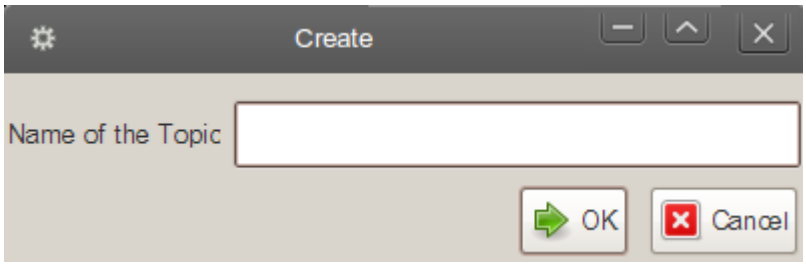


Figura 47: formulari de creació de destinacions

consulta de missatges en cues

Aquesta funcionalitat està únicament disponible per les cues fent *doble-click* damunt d'una d'aquestes en la taula principal o mitjançant el menú contextual de la taula principal. La finestra de consulta de missatges està formada per dues parts, una que enumera els missatges que i són dintre de la cua, i l'altre els detalls del missatge seleccionat. La classe *BrowseMessagesForm* és l'encarregada de representar i gestionar la consulta de missatges. Aquesta trucarà al mètode *browse* de l'administrador de la sessió, recuperant així una llista de JMSMessageInfo per tal d'omplir la taula que enumera els missatges:

```
...
   synchronized (GUI.getSession().getConnection().getAdmin()) {
      JMSMessages = GUI.getSession().getConnection().browse(destName);
    }
   llenaTabla();
...
```

Per tal de mostrar la informació del missatge a la part inferior de la finestra quan es selecciona un missatge, s'utilitza una altra classe que emmagatzema els esdeveniments del formulari de consulta de missatges; *BrowseMessageFormEvents*. S'implementen dos accions

amb el ratolí per la finestra de consulta de missatges, la primera ja s'havia comentat anteriorment; refresca les dades inferiors de la vista amb el missatge seleccionat. La segona servirà per poder copiar les dades com el contingut del missatge, tot fent *doble-click* en la taula inferior damunt la fila del elements corresponent del missatge seleccionat. Aquesta acció obrirà una finestra amb un camp amb el valor seleccionat per tal de poder copiar el valor:

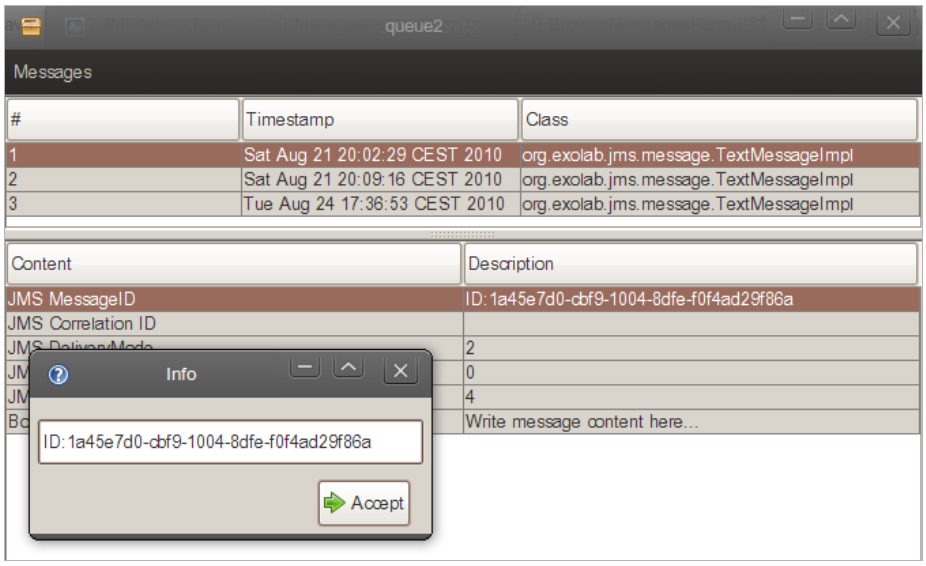


Figura 48: consulta de missatges: selecció de valors

Cal destacar que la taula també mostra el nom de la classe de missatge que implementa el proveïdor JMS gràcies a que la classe *JMSMessageInfo* emmagatzema el nom original d'aquesta.

D'altra banda, s'incorpora la possibilitat d'**exportar els missatges** seleccionats a un directori, generant un fitxer pla de text amb el cos del missatge (sempre que sigui de tipus *TextMessage*). Aquesta opció és disponible mitjançant el menú de la finestra de consulta de missatges. Aquest procés escriu per cada missatge un fitxer amb un nom format pel nom de la cua més la data del missatge.Per imprimir les dades a fitxer, s'utilitza una classe del *package* IO de *JMSMonitor*; *Printer*. El mètode print2File, escriu una cadena de text a un fitxer tot indicant-li el nom:

```
print2File(String path,String filename,String body)
```

6.4.4 Gràfiques

Per tal de dibuixar les gràfiques s'opta per utilitzar *JfreeCharts*, una llibreria de codi obert i gratuïta que permet desenvolupar-les de forma senzilla i que incorpora un conjunt de funcionalitats que la fa força interessant; com l'exportació del gràfic a *png*, la impressió de la gràfica, la modificació de les propietats de la gràfica (colors, nom de les etiquetes...) i gestió d'augment i allunyament dels eixos entre d'altres. Sens dubte es tracta d'una eina que redueix el temps i el cost d'implementació de la funcionalitat de gràfiques amb una gran qualitat i possibilitat de personalització, ja que és de codi obert.

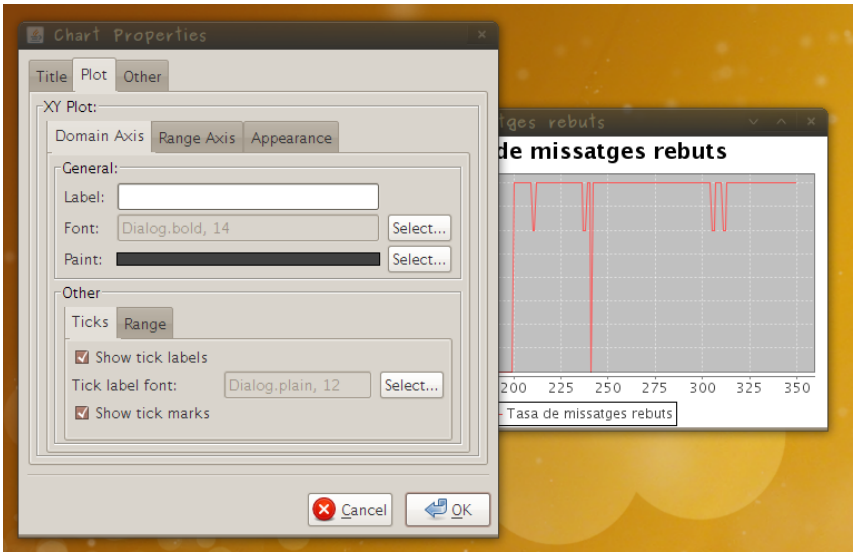


Figura 49: funcionalitats afegides de JFreeCharts

La llibreria implementa molts tipus de gràfiques. El monitor n'utilitzarà dues; una consistent en eixos de coordenades, per representar quantitats en el temps, utilitzada sobretot per monitoritzar missatges entrants i sortints en temps real, i una altra gràfica circular, per comparar diferents paràmetres a la vegada. Aquesta última es mostra en la següent figura:

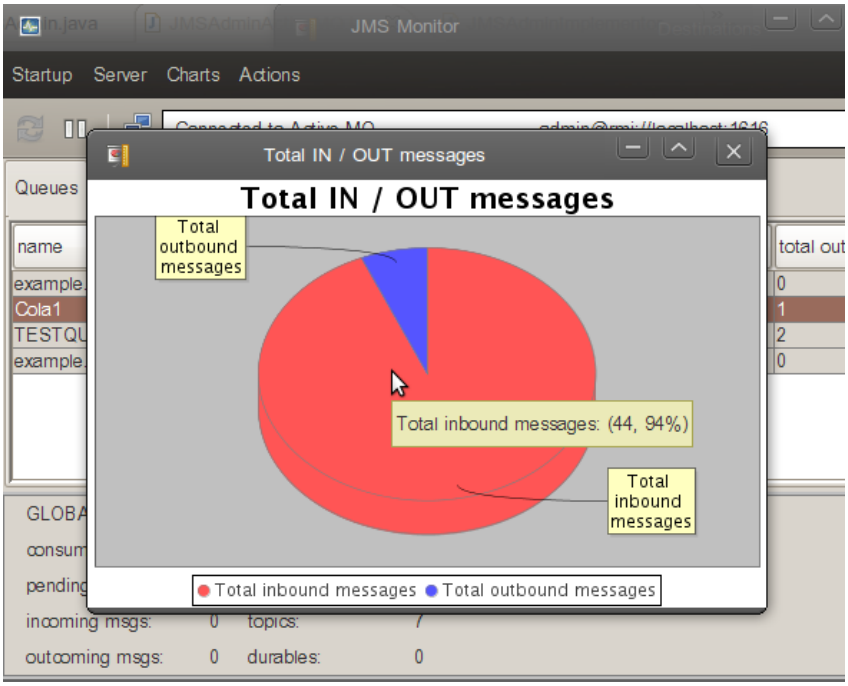


Figura 50: funcionalitats afegides de JFreeCharts

Les dues classes que representen aquestes dues gràfiques en el monitor seran *ChartXYFrame* i *ChartPieFrame,* que implementen la interfície *Charts*:

```
package org.jmsmonitor.gui.frames;
import org.jmsmonitor.gui.elements.ServerInfoModel;

public interface Charts {
   public void refresh(ServerInfoModel s);
   public void destroy();
   public void clean();
}
```

El per què de l'ús d'aquesta interfície és que es simplificarà el codi de la classe principal que controla els diferents objectes i esdeveniments del monitor (*MainFrame*) quan s'hagi de treballar amb gràfiques. Per exemple, quan es crea una gràfica, aquesta s'afegeix a una llista de la classe *MainFrame* que anirà acumulant les referències a objectes que implementen la interfície *Charts*:

```
public class MainFrame extends AbstractFrame implements Observer{
        ...
        private  List <Charts> misCharts;
        ...
}
```

D'aquesta manera quan s'invoqui el mètode *refresh* de la classe *MainFrame* (cal recordar que es trucarà cada vegada que el model de dades de *JMSAdmin* ,"observat" per la classe *MainFrame,* s'actualitzi), es cridarà també al mètode *refresh* de cada objecte que implementi la interfície *Charts,* actualitzant així gràfiques de línies i circulars indistintament:

```
...
if ((this.misCharts!=null)&&(this.misCharts.size()>0)){
        for (int i=0;i<misCharts.size();i++){
             misCharts.get(i).refresh(this.serverInfoModel);
        }
}
...
```

L'usuari per crear una gràfica ho fa mitjançant el menú del monitor. Quan es construeix l'objecte s'indica el tipus de gràfica que serà i s'afegirà a la llista de la classe *MainFrame*, que contindrà totes les gràfiques creades:

```
GUI.addChart((new ChartXYFrame(ChartXYFrame.INBOUNDMESSAGERATE,GUI)));
```

Els tipus d'informació que poden gestionar vénen definits a la pròpia classe com a constants, de tal forma que la gràfica en funció del seu tipus adquireix les dades necessàries per pintar la seva representació:

```
public class ChartXYFrame implements Charts{
        // CONSTANTS
        public static final int INBOUNDMESSAGERATE    = 0;
        public static final int INBOUNDMESSAGETOTAL   = 1;
        public static final int OUTBOUNDMESSAGERATE   = 2;
        public static final int OUTBOUNDMESSAGETOTAL  = 3;
        public static final int QUEUES                = 4;
        public static final int TOPICS                = 5;
        public static final int DURABLES              = 6;
        public static final int CONSUMERS             = 7;
        public static final int PENDINGMSGS           = 8;
```

D'altra banda quan es tanqui una gràfica, aquesta cridarà a la instància de *MainFrame* perquè la tregui de la llista i s'alliberi memòria:

```
  //CLASS XYCHART
  // metode Destroy
 public void destroy(){
      GUI.removeChart(this);
 }

  // CLASS MAINFRAME
  public void removeChart(Charts chart){
          misCharts.remove(chart);
  }
```

6.5 PROVES

Una vegada assolits els objectius plantejats al inici, s'ha realitzat una fase d'anàlisi i proves per tal de verificar les diferents situacions que poden sorgir durant la utilització de l'aplicació. L'estratègia seguida ha sigut la realització de proves funcionals observant i corregint els possibles problemes i desviaments en el desenvolupament.

L'entorn de desenvolupament escollit, *Eclipse*, aporta mecanismes de depuració del codi força útils per trobar i corregir problemes en aquest. A més, l'ús en el projecte de *Log4j*, permet ràpidament detectar un problema en temps d'execució, ja que es van mostrant en consola totes les anotacions en el codi i les excepcions que han aparegut.

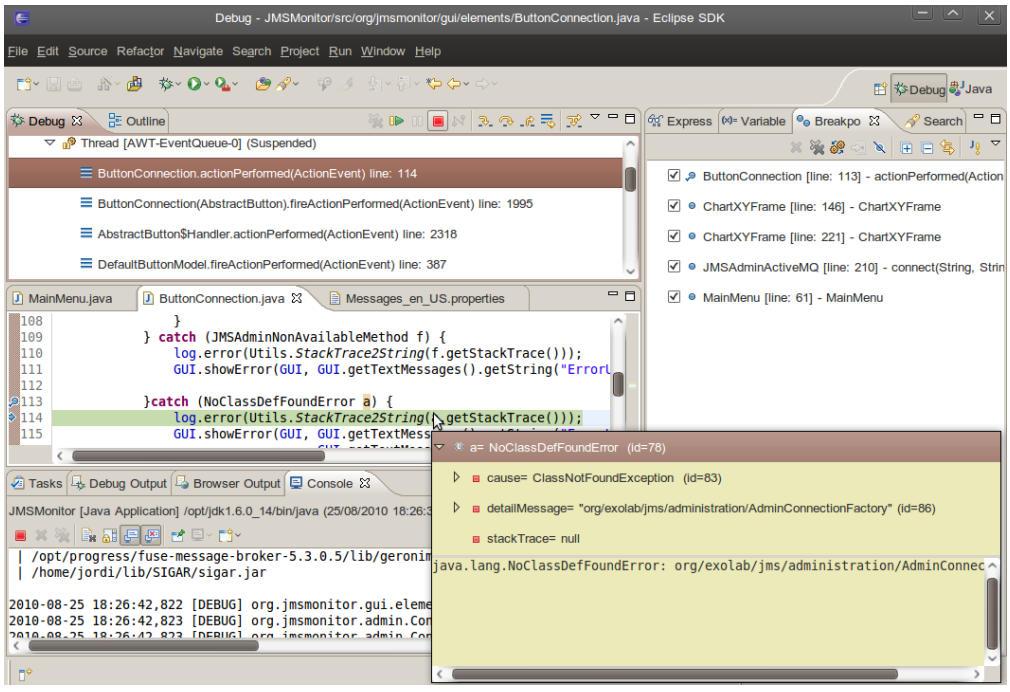


Figura 51: depuració del codi en Eclipse

A continuació es comentaran els problemes més rellevants trobats en les proves funcionals.

Sessions

El formulari en ser acceptat es sotmet a una validació dels seus camps. En concret per validar la direcció es fa servir el mètode *isValidDirection* de la classe *Utils*. El problema és que hi havia protocols no contemplats en un inici que el mètode de comprovació no suportava:

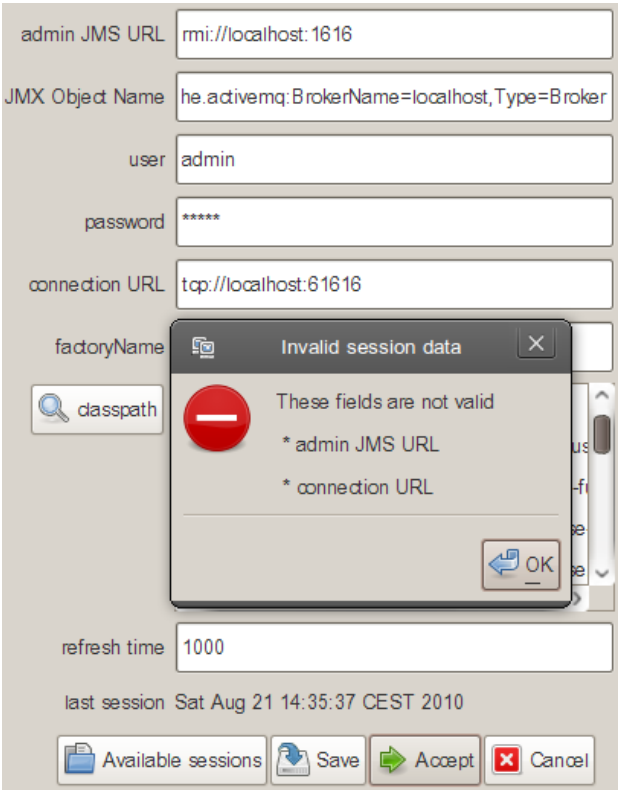


Figura 52: parseig de dades de direccions

Concretament, protocols com *rmi*, *https*, o *tcps* es donaven com invàlids i no es podia carregar la sessió correctament. El mètode de validació en concret feia el següent:

```
try {
      u = new URL(s);
 }catch (MalformedURLException e)
  {
    b = false;
  }
```

S'intentava crear un objecte URL i si ho feia satisfactòriament, es donava aquella direcció com vàlida. No obstant la classe URL només admet aquest protocols; *file, ftp , gopher , http , mailto , appletresource ,doc ,netdoc, systemresource* i *verbatim.* Per tant, basar-se en aquest criteri no és compatible amb la connexió via JMX.

La solució ha sigut modificar el mètode per tal de separar la direcció de connexió en tres parts; protocol, adreça i port, i poder-les validar per separat:

```
    static public boolean isValidDirection(String s)
{
     boolean b = true;
     String aux [] = s.split(":");
     if (aux.length!=3){
           return false;
     }
     if (!isValidProtocol(aux[0])){
            return false;
     }
     if (!isStringValid(aux[1].replace("/", ""))){
            return false;
     }

      String string = aux[2];
      for (int i=0;i<string.length();i++){
          char c=string.charAt(i);
          if (!(c >= '0' && c<='9')){
               return false;
          }
       }
       return b;
      }
```

En primer lloc, el mètode *isValidProtocol* diu si la cadena representa un protocol vàlid (es comparará el paràmetre d'entrada amb una llista de protocols suportats). L'adreça es compara amb un mapa de caràcters limitat (alfanumèric) i per últim, pel port, es recorre cada caràcter de la cadena per verificar si corresponen a nombres reals.

Un altre desviament, es va trobar en la vista. Si es carregava una nova sessió quan una altra ja estava activa i connectada, la interfície es quedava en aquest estat:

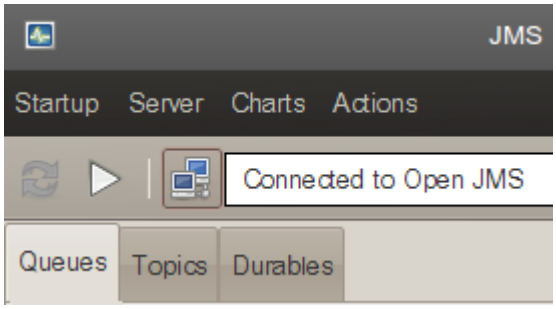


Figura 53: bug en la vista al desconnectar

Aquesta vista no reflexa la situació real: si una sessió ha sigut tancada i s'ha carregat una de nova, no pot aparèixer com connectada. Mirant el mètode de la classe *ManageSessionForm* que tanca la sessió oberta per carregar la nova, es va veure que no s'indica a la vista que es modifiqui després de tancar la sessió:

```
//es tancarà la sessio anterior...
GUI.getSession().close();
if(!GUI.getSession().getName().
                equals(session.getName()))
{
   log.debug("trying to save session to the file");
   //salvar l'ultima data de connexió...
```

Per tant s'afegeix després de tancar la sessió el mètode *setGUItoDisconnect()* que es troba a la classe *ButtonConnection* per a que el botó de connexió canvii el seu estat, i no hi hagi conflictes quan es tingui carregada la nova versió.

Altre problema detectat, es produïa al carregar per segon cop una mateixa sessió ja carregada. Això provoca un *NullPointerException*:

```
   2010-08-25 19:36:54,236 [INFO ]org.jmsmonitor.gui.frames.ManageSessionForm
> Session 'openjms_queue' parsed ok.
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
```

Hi ha un mètode que comprova si la sessió actual és igual a la nova per carregar, de tal forma que si és cert, no es torna a carregar. Aquest mètode anomenat *compare* pertany a la classe *Session* i concretament amb el depurador s'ha pogut comprovar que el *bug* estava en aquesta sentència:

```
if (caux.getObjectName().equals(c.getObjectName()))
```

El problema radicava en que el objecte *Connection* inicialitzava a *null* la variable on s'emmagatzemava el valor a comparar. Així doncs, la correcció consistia en inicialitzar la cadena com buida:

```
//private String  objectName = null;
private String  objectName = "";
```

Consultar missatges

El codi del client gràfic no pot incloure trucades directes a la api JMS, és a dir, sempre ha d'utilitzar la llibreria *JMSMonitorInterface* per tal d'obtenir les dades a monitoritzar. Es va trobar que el codi del formulari de consulta de missatges, treballava amb la classe *Message*, pròpia de la interfície JMS. El problema era que el mètode *browse* de *JMSAdmin* (*JMSMonitorInterface*), retornava una llista de *Message*.

```
abstract List<Message> browseMessages (String url,String
initcontext,String factory, String dest,String user, String pass)
            throws JMSAdminConnectionException, JMSAdminException,
                  JMSAdminNonAvailableMethod,NoClassDefFoundError;
```

Així doncs es va crear una nova classe que emmagatzema aquelles dades importants pel client gràfic dels missatges JMS: *JMSMessageInfo:*

```
public class JMSMessageInfo {

  private String  text;
  private Message message;
  private String  original_classname;
        ...
```

Aquesta oculta l'especificació JMS al codi del client i guarda el nom de la classe original del proveïdor que implementa la interfície *Message* de l'estàndard JMS. Per tant a *BrowseMessageForm*, el formulari de consulta de missatges en cues, es canvia el mètode per directament utilitzar aquesta nova classe de *JMSMonitorInterface*:

```
if (type==MainTableModel.QUEUE){
    synchronized (GUI.getSession().getConnection().getAdmin()) {
            JMSMessages = GUI.getSession().getConnection().browse(destName);
    }
 ...
```

Excepcions

Quan s'estableix una sessió, es necessita tenir carregades les llibreries originals del proveïdor per tal de que *JMSMonitorInterface* pugui utilitzar els seus connectors per adaptar el codi d'aquestes API a una interfície genèrica que entengui el client monitor. Si un usuari no carrega

una llibreria del proveïdor utilitzada pel connector, es llençarà l'excepció *NoClassDefFoundError*. Com que en un inici, durant el desenvolupament es tenien totes les llibreries carregades en el *classpath* del projecte d'*Eclipse*, no es va tenir en compte aquesta excepció.

Per solucionar-lo es va retocar el codi de *JMSMonitorInterface* i *JMSMonitor* per a contemplar-la:

```
//JMSMONITORINTERFACE
//JMSAdminAbstraction
abstract void connect (String url, String user, String pass) throws
JMSAdminConnectionException, JMSAdminNonAvailableMethod,NoClassDefFoundError;
```

```
//JMSMONITOR
//ButtonConnection
try{
     GUI.getSession().getConnection().connect();
    ...
   }catch (NoClassDefFoundError a) {
        log.error(Utils.StackTrace2String(a.getStackTrace()));
         ...
   }
```

Ara doncs, s'informarà al usuari de que una de les classes no s'ha pogut trobar i es mostrarà quina és, per tal de que es pugui cercar quina és la llibreria que falta per incloure en el *classpath*.

Enviament de missatges

Es va detectar, que si s'accedia al formulari mitjançant el menú contextual dintre de la taula de tòpics, i en el mateix formulari es canviava la destinació per que fos a una cua, el missatge s'estava enviant a un tòpic amb el nom de la cua escollida:

```
2010-08-25 18:54:36,447 [DEBUG] org.jmsmonitor.gui.frames.SendMSGForm >
host (tcp://localhost:3035) ,initContext
(org.exolab.jms.jndi.InitialContextFactory) ,factory (QueueCF) ,dest
(queue1) ,body (Write message content here...) ,name(admin) , pass
(admin) ,type(TOPIC)
```

Gràcies al *log* es va veure la part del codi que estava errònia com es mostra en l'anterior imatge. La crida al mètode d'enviament de missatges era incorrecte i es soluciona afegint a la trucada el valor del botó selector de tòpics:

```
c.getAdmin().sendTextMessage(
((c.getJNDIServer()==null||c.getJNDIServer().equals(""))?
                                    c.getHost():c.getJNDIServer()),
c.getCurrentInitContext(),c.getCurrentFactory(),
(String)comboDest.getSelectedItem(), text.getText(), c.getName(),
c.getPass(), (rbuttonTopics.isSelected()));
```

És a dir, el paràmetre del mètode *sendTextMessage isTopic* agafaria el valor de veure si aquest està seleccionat o no.

Interfície JMSMonitorInterface

Amb el connector d'*OpenJMS*, si s'intentava accedir a una *factory* de cues quan s'estava connectat a una de tòpics s'aixecava la següent excepció:

```
  2010-08-25 19:46:13,452 [INFO ] org.jmsmonitor.gui.frames.SendMSGForm > Send
                                  request to purge durable sub2:topic1
Exception in thread "AWT-EventQueue-0" java.lang.ClassCastException:
org.exolab.jms.client.JmsQueueConnectionFactory cannot be cast to
javax.jms.TopicConnectionFactory at
org.jmsmonitor.bridge.plugins.JMSAdminOpenJMS.purge(JMSAdminOpenJMS.java:361)
at org.jmsmonitor.bridge.main.JMSAdmin.purge(JMSAdmin.java:181)
```

El conflicte és que no es pot convertir una *QueueConnectionFactory* (amb la que s'està connectat) en una *TopicConnectionFactory*:

```
TopicConnectionFactory factory = (TopicConnectionFactory) context
                                    .lookup(factoryName);
```

Per tant s'opta per informar a l'usuari de que no es pot accedir al recurs tot indicant l'excepció:

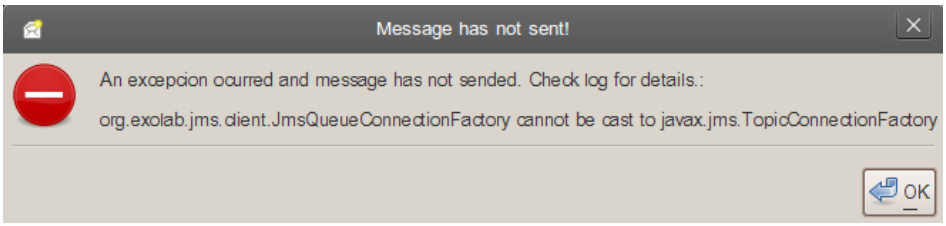


Figura 54: missatge d'error al enviar missatges

En el codi del connector de *OpenJMS*, s'afegeix un bloc de tractament per l'excepció *ClassCastException* i es retorna el missatge per mostrar-lo al usuari com es veu en l'anterior figura:

```
   ...
   } catch (ClassCastException e){
  throw new JMSAdminException (e.getLocalizedMessage());
} ...
```

Al codi de client, s'afegeixen els canvis necessaris per a que el missatge es mostri amb la causa del problema:

```
} catch (JMSAdminException e) {
    log.error(Utils.StackTrace2String(e.getStackTrace()));
    GUI.showError(frame,GUI.getTextMessages().getString("ErrorSend"),
                  GUI.getTextMessages().getString("ErrorSendDescription")
                                + ":\n" + e.getLocalizedMessage());
  }
```

Cal dir que aquesta modificació afecta al formulari de neteja de destinacions i consulta de missatges.

CAPÍTOL 7 Conclusions

7.1 OBJECTIUS ASSOLITS

Com s'ha anat veient al capítol de la implementació del projecte, els objectius plantejats en les pàgines inicials de la present memòria s'han anat assolint.

Des del naixement de l'idea, es tenia molt clar que s'havia de separar al màxim la part gràfica de les diferents implementacions de J*ava Message Service*. Realitzar una llibreria que doni abast a un nombre gran de proveïdors JMS era un repte que no es podia desenvolupar amb el nombre d'hores destinat al projecte, per tant, realitzant el disseny de la llibreria i implementant dos connectors és el primer pas per a que en funció de les necessitats del usuaris, es puguin fer ampliacions en aquesta sense tenir un gran impacte en la part gràfica.

Durant les primeres proves es va poder comprovar que la manera de plantejar la interfície és realment funcional i ràpida d'utilitzar. Aquesta simplicitat fa que en aquest aspecte sigui més

amigable que l'aplicació que ha sigut referent durant les fases de l'elaboració d'aquest programari; *HermesJMS*, encara que aquest és un projecte més gran i més consolidat i amb una compatibilitat entre diferents proveïdors excel·lent. Per tant, un dels objectius que més es volia realçar es pot dir que s'ha acomplert de forma satisfactòria. Cal dir però, que és una eina molt focalitzada a un tipus de tecnologia, i la desconeixença d'alguns conceptes que potser un usuari que realitza test d'aplicacions no ha de conèixer, fan que pugui tenir una certa dificultat per configurar una connexió de forma correcta.

Actualment és una eina que s'està utilitzant per controlar el contingut de cues i enviar missatges a destinacions en un projecte real, que utilitza la tecnologia JMS per integrar un nou component amb la resta de mòduls de l'empresa. Fins a dia d'avui, està facilitant la tasca a l'equip de suport i és un recurs que mica en mica es fa indispensable en el dia a dia.

7.2 DESVIACIONS

En aquest apartat es descriuen les desviacions produïdes en el projecte, tot indicant el motiu i la incidència en l'augment del cost final.

Tasques	Hores Pressupost.	Hores Extra	Total
Anàlisi de requeriments	**31**	**0**	**31**
Implementació	**171**	**23**	**194**
Configuració de l'entorn de desenvolupament	2	5	7
Disseny	**65**	**0**	**65**
Codificació	**104**	**18**	**122**
Implementació de llibreria per ActiveMQ (Fuse)	12	10	22
Funcionalitats amb destinacions i misstages	12	8	20
Avaluació	**31**	**0**	**31**
Documentació	**69**	**8**	**77**
Memòria	61	8	69
TOTAL	**302**	**31**	**333**

Taula 19: resum del cost de les desviacions

Seguint l'anterior taula, es va destinar en un inici poc temps a la configuració de l'entorn de desenvolupament. En la pràctica, el fet de desconèixer el funcionament i l'administració mínima pels dos servidors JMS que s'han utilitzat en el projecte ha supossat un esforç extra. A més, en el cas d'*ActiveMQ*, quan s'estava en la fase d'implementació del codi, es va veure que les funcions per obtenir la informació de destinacions i missatgeria que proporcionava la API d'aquest requerien de l'ús de JMX, una tecnologia desconeguda fins a aquell moment. Així doncs, les hores per la implementació de la interfície van ser superades gairebé amb el doble (10 hores més).

D'altra banda, les funcionalitats definides inicialment de creació i destrucció de destinacions no es van tenir en compte, ja que la eina com bé s'ha anat comentant al llarg de la memòria, estava destinada a la interacció amb missatges i l'obtenció de dades estadístiques per usuaris amb rols de desenvolupament i proves que utilitzin la tecnologia JMS. No obstant, després d'una

primera llista de funcionalitats tancada i pressupostada, es va decidir incloure aquestes ja que poden ser útils també per aquests perfils; en total es van emprar vuit hores més.

En la redacció de la memòria, no es va reflexar el cost que tindria l'apartat de fonaments teòrics. Degut a que la lògica de negoci de l'aplicació és treballar amb un tecnologia concreta, calia explicar amb una mica més de detall la *Java Message Service* i què és el programari d'intermediari.

A més, la falta d'experiència en desenvolupament Java i aquest tipus de solucions, van accentuar més el cost en hores del projecte. A continuació es mostra el cost econòmic total:

RECURS	COST	HORES
Analista	**2.970,00 €**	**165**
Descripció del projecte	54,00 €	3
Estudi d'alternatives	72,00 €	4
Anàlisis de viabilitat	360,00 €	20
Definició necessitats reals	72,00 €	4
Interficie gràfica	432,00 €	24
Gestor de connexions	270,00 €	15
Interficie amb els proveïdors JMS	144,00 €	8
Funcionalitats amb destinacions i missatges	180,00 €	10
Gràfiques	144,00 €	8
Memòria	**1.242,00 €**	**69**
Tècnic programador	**1.570,00 €**	**157**
Configuració de l'entorn de desenvolupament	**70,00 €**	**7**
Gestor de connexions	200,00 €	20
Interficie amb proveïdors JMS	100,00 €	10
Implementació de llibreria per OpenJMS	140,00 €	14
Implementació de llibreria per ActiveMQ (Fuse)	**220,00 €**	**22**
Gràfiques	120,00 €	12
Funcionalitats amb destinacions i missatges	**200,00 €**	**20**
Interficie gràfica	240,00 €	24
Correcció d'incidencies	100,00 €	10
Millores disseny / usabilitat	100,00 €	10
Annexos (documentació)	80,00 €	8
Provador	**99,00 €**	**11**
Proves locals	54,00 €	6
Proves finals	45,00 €	5
TOTAL COSTOS RRHH FINAL		**4.639,00 €**
TOTAL COSTOS RRHH ESPERAT		4.265,00 €
DIFERÈNCIA		374,00 €

Taula 20: costos RRHH finals

7.3 LÍNIES DE DESENVOLUPAMENT OBERTES

Aquesta primera versió de *JMSMonitor* implementa funcionalitats bàsiques que ajuden a programadors, equips de proves i en definitiva, a tots aquells perfils que treballen amb tecnologia JMS. No obstant, durant la implementació s'han anant observant certs aspectes que es podrien millorar i ampliar.

Un dels temes interessants seria la gestió de la càrrega i descàrrega de classes de forma dinàmica en temps d'execució. *JMSMonitor* és una aplicació dissenyada per treballar amb un gran nombre de proveïdors JMS, no obstant es necessita que es carreguin les llibreries dels proveïdors per tal de poder monitoritzar el servidors concrets. En aquesta versió s'inclou la possibilitat de afegir llibreries en forma de fitxers *jar* en temps d'execució al *classpath*, però és un primer plantejament bàsic; millorar la gestió de les classes necessàries per la monitorització dels servidors implementant un sistema per carregar o descarregar-les seria una bona millora per l'aplicació.

S'ha de recordar que només s'han implementat dos connectors pels proveïdors *ActiveMQ* i *OpenJMS*. Ampliar la llibreria *JMSMonitorInterface* amb més connectors potenciaria l'eina sense canvis significatius en el client gràfic. A més, ja es va comentar els diferents problemes amb *OpenJMS* per obtenir informació com per exemple el ràtio de missatges entrants i sortints del servidor. La revisió del connector i la cerca de alternatives per solventar aquest fet podria completar la funcionalitat de la llibreria i per tant seria un valor afegit al programa.

D'altra banda, l'aspecte del client gràfic sempre pot ser millorable i es pot optimitzar el comportament general de la vista, afegint nous menús, nous accessos directes a funcionalitats o fins i tot crear noves gràfiques que comparin diferents paràmetres al mateix temps.

7.4 VALORACIÓ PERSONAL

Amb el desenvolupament de *JMSMonitor* s'ha aconseguit millorar els coneixements personals sobre Java i programari d'intermediari orientat a missatgeria. Motivat per una situació del món laboral, es va decidir cobrir la necessitat de tenir una eina fàcil d'utilitzar que ajudés a realitzar la feina diària respecte a la missatgeria entre aplicacions, creant una aplicació ampliable i multiplataforma.

L'anàlisi i el disseny, tot cercant diferents solucions, proveïdors i llibreries per desenvolupar l'aplicació sens dubte és una de les parts que més treball va portar. Encara que el *planning* d'hores i treball que es va fer en un inici no s'ha complert pel temps setmanal del que al final es disposava per dedicar al projecte, s'ha pogut alliberar en el temps requerit i amb una funcionalitat inicial més que suficient pels objectius establerts.

Amb la implementació, s'han pogut conèixer llibreries i altres tecnologies no directament relacionades amb el projecte, però que ajuden a millorar el desenvolupament i personalment es tindran en compte en un futur. *Log4j* ha sorprès per la seva senzillesa i facilitat d'ús, i com afegeix flexibilitat a l'hora de gestionar els *logs* només modificant un fitxer. D'altra banda, l'elecció de *JfreeCharts* com a lllibreria per generar gràfiques ha sigut més que encertada, aporta molta funcionalitat extra (exportació, zoom...) amb poques hores d'aprenentatge i desenvolupament.

JMSMonitor ha suposat la realització d'un projecte amb totes les seves etapes; anàlisi, viabilitat, planificació i desenvolupament, sent així una experiència enriquidora a l'hora d'iniciar-se en projectes i desenvolupament de programari. Es cert però, que hi ha aspectes que amb més temps hagués estat interessant millorar. No obstant el resultat global és satisfactori, perquè personalment no es tenien precedents semblants a l'actual implementació i l'aprenentatge i fonaments adquirits durant l'elaboració d'aquesta eina han ajudat enriquir els coneixements sobre les tecnologies emprades.

BIBLIOGRAFIA

Monson-Haefel, Richard; Chapell, David A. (2001).*Java Message Service*. O'Reilly.

Snyder Bruce; Bosanac Dejan; Davies Rob. (2008). *ActiveMQ in action*. Manning.

Grant, Scott; P. Kovacs,Michael; Kunnumpurath, Meeraj ; Maffeis, Silvano; Morrison, K. Scott; Suresh Raj , Gopalan; Giotta,Paul; McGovern James. (2000). *Professional JMS Programming*. Wrox Press.

Burd, Barry. (2005). *Eclipse for Dummies*. Wiley Publishing.

Sullins, Benjamin G.;Whipple, Mark B. (2002). *JMX in Action*. Manning.

Freeman, Eric T; Robson,Elisabeth; Bates Bert; Sierra Kathy. (2004). *Head First Design Patterns*. O'Reilly.

Cooper, Mendel. (2010). *Advanced Bash-Scripting Guide* [Documentació en línia].
<http://tldp.org/LDP/abs/html/>

Allen, William; Allen, Linda; (2008). *MS-DOS/MSDOS Batch Files: Batch File Tutorial and Reference.* [Documentació en línia].
<http://www.allenware.com/icsw/icswidx.htm>

RatCliff, Rob. *Intro to CORBA* [Documentació en línia].
<www.futuretek.com/presents/corba/IntroToCorba.pdf>

Recursos en línia

Oracle. PATH and CLASSPATH (The Java™ Tutorials > Essential Classes > The Platform Environment).
<http://download.oracle.com/javase/tutorial/essential/environment/paths.html>

DIA – Gnome Live!
<http://live.gnome.org/Dia>

Eclipse.org home
<http://www.eclipse.org/>

Hermes JMS - Confluence
<http://www.hermesjms.com/confluence/display/HJMS/Home>

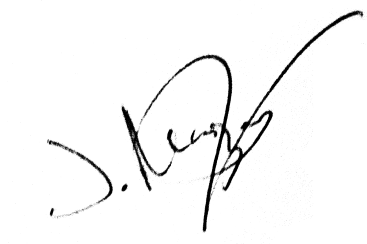
OpenJMS
<http://openjms.sourceforge.net>

Apache ActiveMQ
<http://activemq.apache.org/>

Apache Logging Services Project
<http://logging.apache.org/log4j/>

JfreeChart
<http://www.jfree.org/jfreechart/>

Trail: Creating a GUI With JFC/Swing
<http://download.oracle.com/javase/tutorial/uiswing/index.html>



Jordi Manzano Ulloa

Sabadell, Setembre 2010