SYSTEME ET INFRASTRUCTURE

Middleware Orientés Messages

Principes, mise en oeuvre et outils open source

Smile
OPEN SOURCE SOLUTIONS

www.smile.fr • +33 (0)1 41 40 11 00 • contact@smile.fr
www.smile-oss.com • blog.smile.fr • twitter: @GroupeSmile

PRÉAMBULE

Smile

Smile est une société d'ingénieurs experts dans la mise en œuvre de solutions open source et l'intégration de systèmes appuyés sur l'open source. Smile est membre de l'APRIL, l'association pour la promotion et la défense du logiciel libre, de Alliance Libre, PLOSS, et PLOSS RA, des associations clusters régionaux d'entreprises du logiciel libre.

Smile compte 480 collaborateurs en France, 600 dans le monde, ce qui en fait la première société en France spécialisée dans l'open source.

Depuis 2000, environ, Smile mène une action active de veille technologique qui lui permet de découvrir les produits les plus prometteurs de l'open source, de les qualifier et de les évaluer, de manière à proposer à ses clients les produits les plus aboutis, les plus robustes et les plus pérennes.

Cette démarche a donné lieu à toute une gamme de livres blancs couvrant différents domaines d'application. La gestion de contenus (2004), les portails (2005), la business intelligence (2006), les frameworks PHP (2007), la virtualisation (2007), et la gestion électronique de documents (2008), ainsi que les PGIs/ERPs (2008). Parmi les ouvrages publiés en 2009, citons également « Les VPN open source », et « Firewall est Contrôle de flux open source », et « Middleware », dans le cadre de la collection « Système et Infrastructure ».

Chacun de ces ouvrages présente une sélection des meilleures solutions open source dans le domaine considéré, leurs qualités respectives, ainsi que des retours d'expérience opérationnels.

Au fur et à mesure que des solutions open source solides gagnent de nouveaux domaines, Smile sera présent pour proposer à ses clients d'en bénéficier sans risque. Smile apparaît dans le paysage informatique français comme le prestataire intégrateur de choix pour accompagner les plus grandes entreprises dans l'adoption des meilleures solutions open source.

Ces dernières années, Smile a également étendu la gamme des services proposés. Depuis 2005, un département consulting accompagne nos clients, tant dans les phases d'avantprojet, en recherche de solutions, qu'en accompagnement de projet. Depuis 2000, Smile dispose d'un studio graphique, devenu en 2007 Smile Digital – agence interactive, proposant outre la création graphique, une expertise e - marketing, éditoriale et interfaces riches. Smile dispose aussi d'une agence spécialisée dans la TMA (support et l'exploitation des applications) et d'un centre de formation complet, Smile Training. Enfin, Smile est implanté à Paris, Lille, Lyon, Grenoble, Nantes, Bordeaux, Poitiers, Aix-en-Provence et Montpellier. Et présent également en Espagne, en Suisse, au Benelux, en Ukraine et au Maroc.

Quelques références

Intranets et Extranets

Société Générale - Caisse d'Épargne - Bureau Veritas - Commissariat à l'Energie Atomique - Visual - CIRAD - Camif - Lynxial - RATP - Sonacotra - Faceo - CNRS - AmecSpie - INRA - CTIFL - Château de Versailles - Banque PSA Finance - Groupe Moniteur - Vega Finance - Ministère de l'Environnement - Arjowiggins - JCDecaux - Ministère du Tourisme - DIREN PACA - SAS - CIDJ - Institut National de l'Audiovisuel - Cogedim - Diagnostica Stago Ecureuil Gestion - Prolea - IRP-Auto - Conseil Régional Ile de France - Verspieren - Conseil Général de la Côte d'Or - Ipsos - Bouygues Telecom - Prisma Presse - Zodiac - SANEF - ETS Europe - Conseil Régional d'Ile de France - AON Assurances & Courtage - IONIS - Structis (Bouygues Construction) - Degrémont Suez - GS1-France - DxO - Conseil Régional du Centre - Beauté Prestige International - HEC - Veolia

Internet, Portails et e-Commerce

Cadremploi.fr - chocolat.nestle.fr - creditlyonnais.fr - explorimmo.com - meilleurtaux.com - cogedim.fr - capem.fr - Editions-cigale.com - hotels-exclusive.com - souriau.com - pci.fr - odit-france.fr - dsv-cea.fr - egide.asso.fr - Osmoz.com - spie.fr - nec.fr - vizzavi.fr - sogeposte.fr - ecofi.fr - idtgv.com - metro.fr - stein-heurtey-services.fr - bipm.org - buitoni.fr - aviation-register.com - cci.fr - eaufrance.fr - schneider-electric.com - calypso.tm.fr - inra.fr - cnil.fr - longchamp.com - aesn.fr - bloom.com - Dassault Systemes 3ds.com - croix-rouge.fr - worldwatercouncil.org - Projectif - credit-cooperatif.fr - editionsbussiere.com - glamour.com - nmmedical.fr - medistore.fr - fratel.org - tiru.fr - faurecia.com - cidil.fr - prolea.fr - bsv-tourisme.fr - yves.rocher.fr - jcdecaux.com - cg21.fr - veristar.com - Voyages-sncf.com - prismapub.com - eurostar.com - nationalgeographic.fr - eau-seine-normandie.fr - ETS Europe - LPG Systèmes - cnous.fr - meddispar.com - Amnesty International - pompiers.fr - Femme Actuelle - Stanhome-Kiotis - Gîtes de France Bouygues Immobilier - GPdis - DeDietrich - OSEO - AEP - Lagardère Active Média - Comexpo - Reed Midem - UCCIFE - Pagesjaunes Annonces - 1001 listes - UDF - Air Pays de Loire - Jaccede.com - ECE Zodiac - Polytech Savoie - Institut Français du Pétrole - Jeulin - Atoobi.com - Notaires de France - Conseil Régional d'Ile-de-France - AMUE

Applications métier

Renault - Le Figaro - Sucden - Capri - Libération - Société Générale - Ministère de l'Emploi - CNOUS - Neopost - Industries - ARC - Laboratoires Merck - Egide - ATEL-Hotels - Exclusive Hotels - CFRT - Ministère du Tourisme - Groupe Moniteur - Verspieren - Caisse d'Epargne - AFNOR - Souriau - MTV - Capem - Institut Mutualiste Montsouris - Dassault Systèmes - Gaz de France - CAPRI Immobilier - Croix-Rouge Française - Groupama - Crédit Agricole - Groupe Accueil - Eurordis - CDC Arkhineo

Applications décisionnelles

IEDOM – Yves Rocher - Bureau Veritas - Mindscape – Horus Finance – Lafarge – Optimus – CecimObs – ETS Europe – Auchan Ukraine – CDiscount – Maison de la France – Skyrock – Institut National de l'Audiovisuel – Pierre Audouin Consultant – Armée de l'air – Jardiland – Saint-Gobain Recherche – Xinek – Projectif – Companeo – MeilleurMobile.com – CG72 – CoachClub

Ce livre blanc

Les Middleware Orientés Messages, ou « MOMs », sont des outils particulièrement précieux pour mettre en œuvre des échanges entre applications de toutes natures. Comme il arrive très souvent dans ce qui touche aux infrastructures, les solutions open source sont particulièrement en pointe dans ce domaine. Parce que le middleware est souvent le ciment de toute une architecture, les critères d'ouverture, de pérennité et d'indépendance sont essentiels dans le choix d'un tel outil, et personne ne souhaite dépendre, dans ce contexte, de la politique commerciale de tel ou tel acteur particulier.

C'est pourquoi les solutions open source sont en position de force en matière de middleware. La force de l'open source, c'est aussi la diversité et le foisonnement de l'offre, dans une dynamique de compétition qui fait naître des produits de grande qualité. C'est le cas en matière de MOM, où il existe différentes solutions tout à fait solides et matures.

Ce livre blanc vise à présenter l'offre open source en matière de MOM. Nous avons identifié quatre solutions qui se distinguent par leur qualité, leur robustesse et la stature de leur éditeur.

Après avoir présenté les concepts fondamentaux et les fonctionnalités communes à tous ces outils, nous étudierons chacun d'eux de manière plus détaillée.

Sommaire

CONCEPTS DES MOMS ET JMS

Qu'est-ce qu'un Middleware ?

Un middleware est un logiciel qui permet à différentes applications d'échanger et d'interopérer.

Un middleware permet aux applications d'interopérer y compris lorsqu'elles tournent sur des serveurs différents, interconnectés par un réseau. Le middleware est un outil de haut niveau, puisqu'il offre ses services aux *applications*, mais les échanges induits s'appuient sur toute une pile de protocoles réseau.

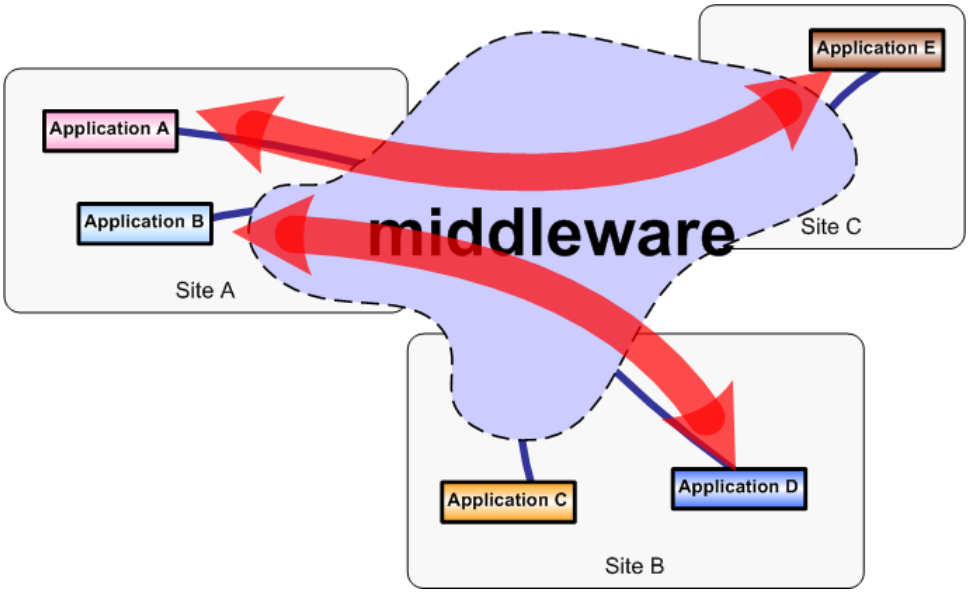
Par exemple, les outils qui permettent à des applications d'invoquer les services d'un SGBD sont une catégorie particulière de middlewares.

Parmi les middlewares qui permettent l'interopérabilité entre applications homologues (de même nature), on peut distinguer deux grandes familles:

- Les middlewares qui permettent l'invocation synchrone de fonctions et méthodes, parmi lesquels on trouve la famille des *request brokers*, avec CORBA ou encore DCOM.

- Les middlewares d'échange asynchrones, qui sont principalement à base de messages, ce sont les MOMs, les *Message Oriented Middleware.*

Un middleware est davantage qu'un simple protocole d'appel des services offerts par une application, et typiquement RPC, RMI ou bien SOAP, tous également *synchrones,* ne sont pas vraiment considérés comme des middlewares.



Outre la gestion de l'échange proprement dit, les services offerts par un middleware peuvent être de différentes natures, en particulier:

- *L'identification* et la *localisation* des applications à un niveau supérieur, au dessus des adresses réseau et des noms de serveurs, et l'acheminement des échanges à ce niveau.

- Dans certains cas, la *conversion de formats* de représentation des données entre les applications, permettant à des applications d'environnements et langages différents d'échanger de manière transparente.

- Dans certains cas également, des fonctions de *sécurité*, de *répartition de charge* ou de gestion du *secours*.

Pourquoi des échanges asynchrones ?

Lorsqu'une application invoque les services d'une autre application au moyen d'un middleware synchrone, il faut impérativement :

- que la seconde application soit en état de marche, à l'instant où elle est invoquée ;

- qu'elle soit joignable par le réseau.

Si l'une ou l'autre de ces conditions n'est pas réunie, la première application doit renoncer à invoquer le service distant. Dans certains cas, cette impossibilité peut avoir des conséquences graves pour l'application initiatrice de l'échange, qui doit être prévue pour traiter l'échec de l'appel. L'invocation synchrone d'un service distant crée une dépendance très forte entre les deux applications.

Et quand bien même ces deux conditions sont réunies, la question se pose encore du temps de réponse de cet appel de service. L'application appelante peut-elle rester en attente de la réponse ? Peut-elle faire attendre un utilisateur ? Après combien de temps doit-elle renoncer ?

Dans certains contextes, les échanges synchrones sont possibles. En particulier lorsque les deux applications sont sur le même serveur, ou à la rigueur sur la même plateforme, et que leurs temps de réponse peuvent être garantis.

Dans tous les autres cas, la dépendance qu'implique un mode d'échange synchrone, tant au niveau des applications elles-mêmes que des serveurs, est néfaste.

Au contraire, avec un middleware asynchrone, l'application initiatrice de l'échange ne reste pas en attente d'une réponse : elle confie son message au middleware et poursuit sont traitement.

On dit qu'un middleware asynchrone met en œuvre une faible dépendance, un couplage lâche *(« loose coupling »)*, entre les applications, ce qui permet une bien plus grande flexibilité dans les architectures.

Les Middleware Orientés Messages, ou MOM, sont de loin les implémentations les plus courantes du principe d'échanges asynchrones et, comme nous le verrons, il existe un standard en la matière, la spécification JMS, qui a un bon nombre d'implémentations de qualité.

Les Middlewares Orientés Messages ou MOM

Définition

On l'a vu, les MOMs sont des middlewares, des outils d'échange qui permettent à des applications de communiquer en échangeant des messages. Une application « A » doit adresser un message à une application « B », qui tourne (peut-être) sur un serveur différent. L'application « A » confie son message au MOM, qui se charge de l'acheminer et de le remettre à l'application « B ».

L'objet véhiculé par le MOM entre deux applications est appelé *message*. Mais rien n'est imposé quant à *ce que représente* ce *message*, sa taille, ou encore le format des données qu'il véhicule. Pour l'essentiel, ces questions ne concernent que l'application « A » et l'application « B », qui doivent partager un certain nombre de conventions, afin de se comprendre.

Le MOM, quant à lui, ne s'intéresse donc pas au contenu du message, il ne fait que le transmettre, et il le remet au destinataire sans y avoir apporté de changement.

MOM, EAI, ESB

À la différence d'un MOM, un outil d'EAI (*Enterprise Application Integration*), est aussi en charge de réaliser transformations sur les informations portées par les messages, afin d'adapter les données de l'émetteur aux formats gérés par le destinataire.

Un EAI englobe donc les fonctionnalités du MOM, et y ajoute des possibilités facilitant l'intégration des applications au niveau des données transférées.

Dans un MOM, comme on l' a vu, les applications doivent *parler le même langage*, tandis qu'un EAI au contraire prend en charge les traductions entre représentations différentes.

Un EAI est donc un middleware qui a comme principales fonctions :

- L'interconnexion des systèmes hétérogènes.
- La gestion de la transformation des messages.
- La gestion du routage des messages.

L'ESB, *Enterprise Service Bus*, est un concept plus ambitieux encore, qui se présente comme le socle uniforme d'une architecture SOA globale. Là où l'EAI *peut prendre en charge* des transformations de formats permettant à une application A d'interopérer avec une application B, l'ESB généralise le concept, en posant pour principe qu'*il suffit qu'une application A soit interfacée à l'ESB* pour qu'elle puisse interopérer par son intermédiaire avec toute autre application interfacée à l'ESB. Et par ailleurs, la connexion à l'ESB n'est pas exclusivement à base de messages, elle doit supporter une grande diversité de modes d'échange et de protocoles.

EDA, Event Driven Architecture

Puisque nous évoquons quelques acronymes en vogue, il faut parler aussi du concept EDA, « *Event-Driven Architecture* », architecture pilotée par les événements, qui est à certains égards une alternative à l'approche SOA.

L'approche EDA part de l'idée que tout traitement est d'une certaine manière exécuté en réaction à un événement. Et bien sûr, tout traitement est par ailleurs générateur d'événements. Ainsi, la vente d'un produit est un événement, qui induit un ensemble de traitements relatifs par exemple à la gestion des stocks, à la comptabilité, à la logistique, à la relation client, etc. Tout est événement, tout est réaction à des événements, et il en va de même pour nous-mêmes, êtres humains, qui agissons en réaction à un ensemble de stimuli externes.

Dans l'approche EDA, la réaction à un événement n'est pas un traitement synchrone. Elle peut avoir des exigences de rapidité, mais

elle est par essence asynchrone. Alors que l'approche SOA, même si elle peut se décliner dans une logique asynchrone, est malgré tout par essence une approche synchrone. Et bien sûr, les MOMs sont le support naturel d'une approche EDA.

Un dernier acronyme à trois lettres pour la route: CEP, pour « *Complex Event Processing* », traitement d'événements complexes, consiste à identifier, puis traiter, des événements complexes à partir d'une combinaison d'événements simples. C'est donc un concept complémentaire à l'approche EDA, partant du principe qu'il ne suffit pas de réagir à des événements individuels, il faut être en mesure d'identifier des événements de plus haut niveau, comme résultante d'événements élémentaires. Par exemple: un ordre de vente, plus un autre ordre de vente, plus encore un ordre de vente... égal une crise financière, événement complexe, s'il en est !

Des échanges asynchrones

Les échanges de messages mis en œuvre par les MOMs sont *asynchrones*. Cela signifie que les applications ne sont pas en attente d'une réponse à leur message. En fait, il est possible qu'un message de réponse soit attendu, mais dans ce cas il n'y a pas de délai garanti pour cette réponse, de sorte que l'application ne doit pas se bloquer en attente de la réponse, et encore moins faire attendre un utilisateur. Le caractère asynchrone ne dit rien quant au délai d'acheminement du message : il peut être très rapide, de quelques millisecondes à peine, mais il ne doit pas être considéré comme assuré.

Des échanges fiables

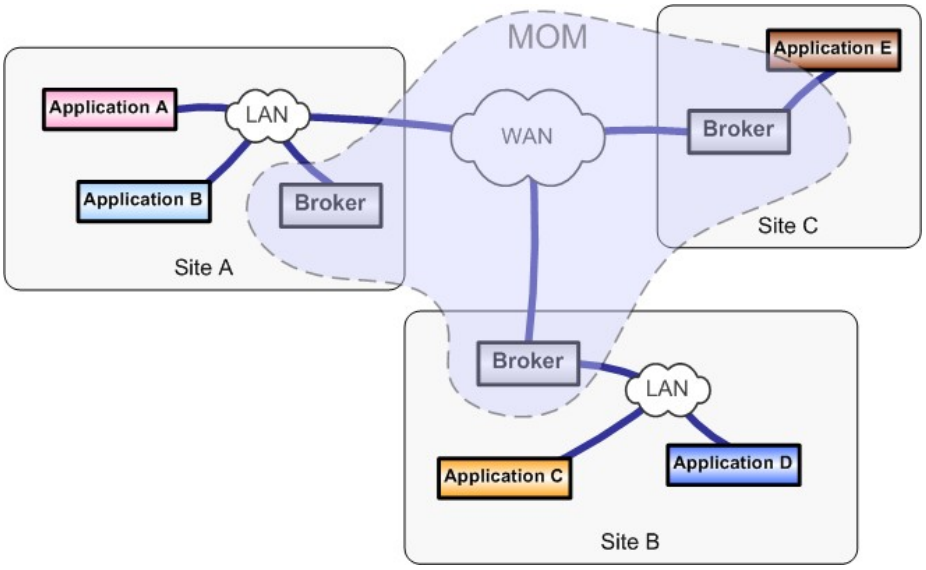
L'une des qualités attendues des MOMs est de garantir l'acheminement des messages *quelles que soient les circonstances, les aléas,* et en particulier y compris dans le cas où la connectivité réseau est interrompue, où le serveur distant est arrêté, ou bien où l'application destinatrice n'est pas en mesure de réceptionner les messages. Dans tous ces cas de figure, le MOM doit conserver les messages qui lui sont confiés jusqu'à ce qu'ils aient été remis, et même, jusqu'à ce qu'ils aient été *correctement traités* par l'application destinatrice.

Nous verrons que cette fiabilité de l'acheminement peut être rendue plus ou moins forte, selon les paramètres et la configuration du MOM.

Les échanges à base de MOM ne sont pas, par nature, en mode *requête / réponse*, comme peut l'être un échange HTTP par exemple. Il est possible bien sûr que l'application destinatrice émette à son tour un message, que l'on peut considérer comme une réponse, mais il s'agit alors seulement d'une utilisation particulière du MOM.

Brokers

Les *brokers* sont des programmes gérant le flux de messages. En d'autres termes, un MOM est composé d'un ou de plusieurs brokers. Comme le montre la figure suivante, c'est avec les brokers que les applications clientes communiquent, au travers de l'API.

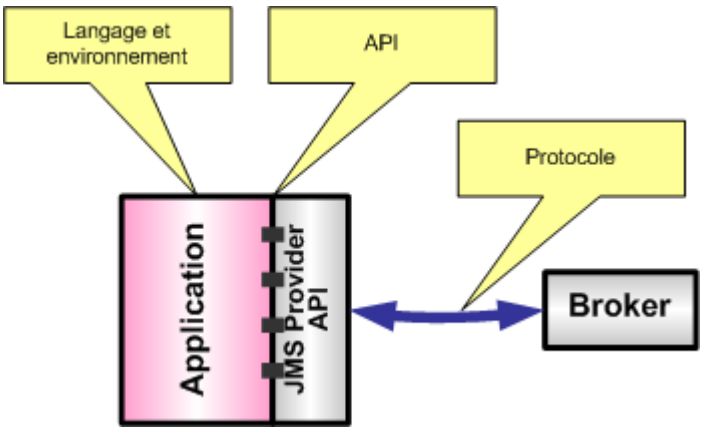


Un broker est un *serveur* au sens logiciel du terme, c'est-à-dire un *processus* qui est à l'écoute des requêtes qui peuvent lui être adressées par d'autres *processus*, les applications clientes.

Une *plateforme MOM* ou plateforme middleware est donc constitué d'un ensemble des *brokers* et des *passerelles*.

Protocoles et APIs

Lorsqu'une application échange avec un broker, par exemple pour lui remettre un message, et de même lorsqu'un broker échange avec un autre broker, ces échanges mettent en œuvre un *protocole réseau*. Le protocole définit les commandes invoquées et leurs paramètres, ainsi que la représentation des données, entêtes et corps, constituant les messages.



Ce protocole est généralement invisible pour les applications, qui ne voient que des appels de fonctions, des APIs. Et de même, pour ce qui est des échanges entre deux brokers d'un même MOM, le protocole peut être considéré comme une affaire privée, interne, relevant purement de l'implémentation du MOM. C'est pourquoi on s'intéresse généralement davantage à l'ouverture des MOMs en termes d'APIs qu'en termes de protocoles d'échange.

Pourquoi un MOM open source ?

Un middleware est nécessairement *structurant* pour les applications qui en font usage, c'est-à-dire que les applications seraient un peu différentes si elles utilisaient un autre middleware, et en conséquence, changer de middleware pourrait impliquer des changements sur toutes les applications, avec donc un coût important.

En conséquence, il est clair que l'on ne souhaite pas avoir à changer de middleware, et qu'il vaut mieux éviter aussi d'avoir un fournisseur en position de tirer profit de cette dépendance.

C'est une des raisons pour lesquelles les solutions open source sont naturellement à privilégier pour cette typologie d'outils.

Et c'est pourquoi aussi les grands acteurs de l'open source ont depuis longtemps placé les middleware au premier rang de leurs priorités, ce qui explique que l'on ait aujourd'hui un large choix de produits de qualité, comme on le verra.

Il faut « rendre à César ce qui appartient à César », et rappeler que le père de tous les MOMs est sans doute le produit MQSeries, de IBM, aujourd'hui renommé « Websphere MQ », un produit introduit dans les années 90, et qui a rencontré un grand succès en particulier dans les banques et autres grands comptes IBM. MQSeries a posé les concepts du MOM, échanges asynchrones et fiables, en offrant par ailleurs des connecteurs pour une diversité d'environnements.

Aujourd'hui, les solutions open source sont en position de force. Elles sont généralement plus respectueuses des standards, plus ouvertes, et – pour certaines d'entre elles au moins – plus dynamiques dans leur développement. Et elles présentent un coût total de possession bien plus avantageux.

Les services d'un MOM

Le service de base d'un MOM est d'acheminer un message d'une application vers une autre.

Mais il a d'autres valeurs ajoutées, d'autres caractéristiques :

Un service fiable

Le MOM garantit à l'application A que le message qui lui est confié ne sera pas perdu. Ceci, même en présence d'incidents de différentes natures (logiciels, matériel, réseau). L'application émettrice *peut compter* sur le MOM, et le fait de pouvoir compter sur lui permet de simplifier la conception de l'application. On peut, à différents égards, faire un parallèle entre un MOM et une base de données. Lorsqu'une application a écrit une donnée dans un SGBD, elle peut compter que cette donnée ne sera pas perdue. Les mécanismes qui permettent d'assurer ceci peuvent être complexes, mais l'application n'a pas à s'en préoccuper. C'est la même chose pour un MOM. Le MOM peut donc être utilisé y compris pour transporter des objets critiques, des transactions financières par exemple. Nous verrons plus loin que l'on peut, dans certains contextes d'utilisation, choisir de se passer de cette fiabilité.

Un service asynchrone

L'application A confie son message au MOM, à destination de l'application B. Mais l'application B est peut-être saturée, ou bien arrêtée, son serveur est peut-être en panne, ou bien injoignable. Rien de tout cela ne pose problème: le MOM attendra. Que le réseau remarche, que le serveur soit en état, que l'application soit lancée. Il attendra jusqu'à avoir pu remettre le message à son destinataire. Et même un peu plus: jusqu'à ce que son destinataire ait indiqué que le message a pu être traité avec succès.

Une indirection de nommage

Nous avons jusqu'ici fait comme si l'application A remettait au MOM un message *« à destination de l'application B »*. Ce n'est pas tout à fait exact, et la nuance est importante. L'application A remet au MOM un message *à destination d'une file d'attente, d'une queue*. Et une application B (mais peut-être aussi différentes applications B1, B2, …) peut lire les messages de cette queue, selon des modalités que nous verrons plus en détail. Cette indirection est importante: l'application A *ne connait pas*

l'application B, ne connait ni son « nom », ni le serveur sur lequel elle tourne, ni dans quel sous-réseau ce serveur peut se trouver. Néanmoins, le message sera remis à l'application B. On voit que le principe de *couplage lâche* n'est pas que dans le caractère asynchrone, il est important également en ce qui concerne l'identification des applications prenant part aux échanges.

Pas de transformation des données

À la différence d'autres middleware, et en particulier la famille des ORB, les MOMs ne prennent pas en charge de transformation de la représentation des données. Le MOM reçoit un message d'une application A, et le remet tel quel, inchangé, à une application B. Les applications échangeant grâce au MOM doivent donc *« parler le même langage », c'est-à-dire représenter leurs objets* (chaînes, nombres, matrices, classes, dates, etc.) *de la même manière, au sein des messages, pour se comprendre*.

Autres services

Nous verrons que la gestion de la répartition de charge et la gestion du secours sont extrêmement faciles à mettre en œuvre au moyen d'un MOM. La possibilité d'avoir plusieurs applications lisant et traitant les messages sur une même queue, implémente une répartition de charge très simple, mais très efficace.

Java Messaging System ou JMS

Introduction

JMS est l'API de *communication asynchrone via Message* de Java. C'est l'API qui permet à une application d'invoquer les services d'un MOM.

JMS fait partie de JEE 5 et est ainsi disponible aux applications tournant sur des serveurs applicatifs Java.

La première version de JMS, JMS 1.0.2b est sortie le 25juinn 2001. La seconde version, JMS 1.1 est sortie le 18 mars 2007, sans présenter de différence importante. Les classes JMS 1.1 permettent de réaliser des clients JMS plus facilement. Nous allons étudier sommairement JMS 1.1. Mais nous commencerons par poser quelques définitions et concepts.

Comme toute spécification, JMS doit assurer que toutes les applications qui s'y conforment ont le même comportement quel que soit le fournisseur de l'implémentation. La JMS laisse aussi dans des cas bien définis, la liberté aux fournisseurs d'implémenter ou non certaines fonctionnalités. Nous reviendrons en détail sur ces fonctionnalités qui distinguent les différents MOMs.

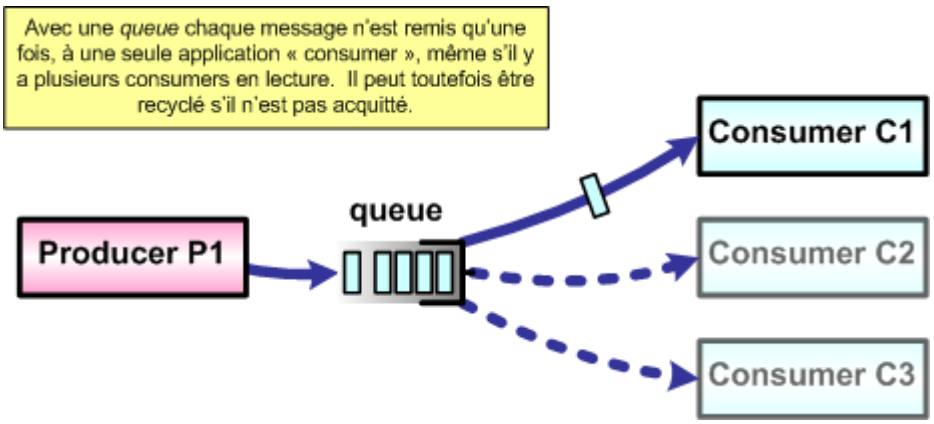
Comme JDBC pour l'accès aux bases de données, ou JCR pour l'accès à un référentiel de contenus, JMS permet en théorie de développer une application interfacée à un MOM, sans dépendre d'un produit particulier. C'est-à-dire qu'il devrait être possible de remplacer un MOM JMS par un autre de manière transparente pour l'application. Comme pour les accès aux bases de données, cet aspect interchangeable n'est pas toujours vérifié en pratique. Il peut exister des petites différences d'implémentation de la spécification, et par ailleurs les différents outils MOMs s'efforcent d'offrir des petits « plus », des fonctionnalités différenciantes.

Modes de communication

La spécification JMS introduit deux modes de communication, les « *domaines JMS* »: les *topics* d'une part, les *queues* d'autre part..

Le mode point à point ou « queue »

Ce mode de communication est aussi appelé communication *via queue*. Une application envoie des messages à une queue. Une seule des applications connectées reçoit le message. Il peut y avoir plusieurs applications en lecture sur la queue, mais une seule d'entre elles recevra le message.



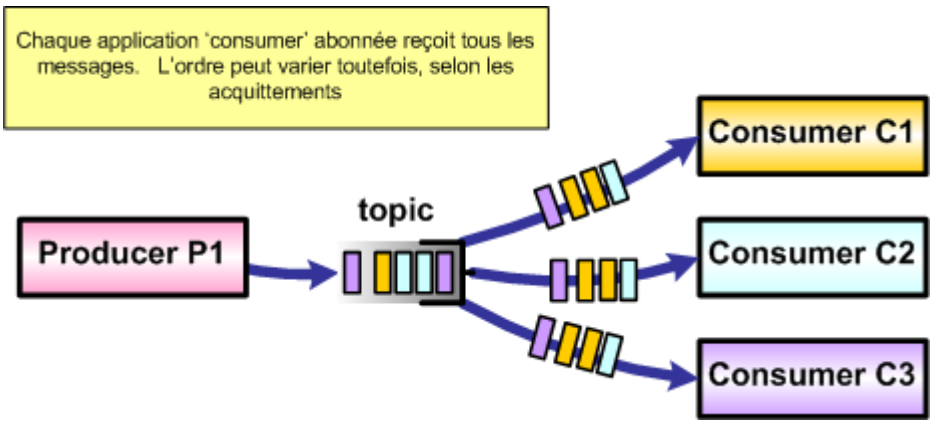
Le mode « *publish-subscribe* » ou « *topic* »

Ce mode de communication est aussi appelé communication *via topic*. Une application envoie des messages à des topic.

Dans ce mode, on dit que les applications *s'abonnent (subscribe)* à un *topic,* afin de recevoir les messages. Plusieurs applications peuvent être abonnées à un même *topic,* et chacune d'elles reçoit une copie des messages.

À la manière de la diffusion d'un magazine par exemple, l'émetteur *publie* un message, et les différents destinataires *s'abonnent* pour recevoir une copie du message.

C'est donc un échange de 1 vers N, mais qui peut être aussi bien « de P vers N », car plusieurs applications peuvent écrire dans le *topic*.



Queues et topics

On voit bien les différences d'usage de ces deux modes. Dans le mode *queue*, on peut imaginer qu'un message représente une *unité de traitement*. L'application destinatrice reçoit le message et effectue un *traitement* à partir du message, et dans ce cas il faut que le traitement ne soit pas exécuté deux fois. Dans le mode *topic*, on peut voir le message plutôt comme une unité *d'information*, qui peut intéresser différents acteurs, différentes applications. Par exemple, un ordre de bourse sera une unité de traitement, tandis qu'un cours de bourse sera une information.

Queue et *Topic* sont regroupés sous le nom de « *Domaine* ». Ainsi, « *envoyer un message à un domaine* » équivaut à « *envoyer un message à une queue ou à un topic* ».

Quelques définitions

JMS introduit différents termes et concepts que nous allons rapidement parcourir:

JMS Client

Un *client JMS* est une application écrite en Java envoyant et/ou recevant des messages au moyen de l'API JMS.

Non-JMS Client

Un *client non-JMS* est une application envoyant et/ou recevant des messages en communiquant avec le JMS Provider selon son protocole particulier, soit en direct, soit par l'intermédiaire des fonctions d'une API. Cette application n'est pas écrite en Java.

JMS Provider

Un *Fournisseur JMS* est une implémentation des services JMS écrite en Java. Ainsi,i les MOMs que nous étudierons plus loin sont des *JMS Providers.*

JMS Consumer

Un *Consommateur JMS* est une application qui reçoit et traite des messages JMS.

JMS Producer

Un *Producteur JMS* est une application qui crée et envoie des messages JMS. Une même application peut être à la fois *JMS Producer* et *Consumer.*

JMS Message

Le *message JMS* est l'unité fondamentale de JMS. Il est envoyé et reçu par des *Client JMS*.

JMS Domains

Les deux *domaines JMS* correspondent aux deux modes de communication déjà évoqués : point à point avec les *queues* ou publish-subscribe avec les *topics*.

Destination

Les objets *destinations* sont des objets servant à identifier la cible des messages à envoyer ou à recevoir, c'est-à-dire des domaines, *queues* et *topics*.

Encodage du Corps des messages

Même si le contenu et le format du corps sont fondamentalement l'affaire des applications, JMS aide les applications à manipuler certains types d'objets en fournissant différents types de corps de message.

Le corps des messages peut être encodé selon les 5 « *Message Types* » disponibles :

- « *TextMessage* » : Le corps contient des caractères.

```
StringstockData;
TextMessage message;
message = session.createTextMessage();
message.setText(stockData);
String stockInfo;  /* String to hold stock info */
stockInfo = message.getText();
```

- « *BytesMessage* » : Le corps contient une suite de bytes, selon le langage Java

```
byte[]stockData;   /* Stock information as a byte array */
BytesMessage message;
message = session.createBytesMessage();
message.writeBytes(stockData);
byte[]stockInfo; /* Byte array to hold stock information */
int   length;
length = message.readBytes(stockData);
```

- « *MapMessage* » : Le corps contient une map. Une map est un type de données reliant une clef (codée en String) a une valeur (codée en String, Double ou Long)

```
message = session.createMapMessage();
/* First parameter is the name of the map element, * second is the value
*/
message.setString("Name", "SUNW");
message.setDouble("Value", stockValue);
message.setLong("Time", stockTime);
message.setDouble("Diff", stockDiff);
message.setString("Info", "Recent server announcement causes market
interest");
stockName = message.getString("Name");
stockDiff = message.getDouble("Diff");
stockValue = message.getDouble("Value");
stockTime = message.getLong("Time");
```

- « *StreamMessage* »

Ce type permet de concaténer plusieurs type natif (String, Double ou Long).

```
/* Create message */ message = session.createStreamMessage();
/* Set data for message */
message.writeString(stockName); message.writeDouble(stockValue);
message.writeLong(stockTime);
message.writeDouble(stockDiff);
message.writeString(stockInfo);

stockName = message.readString();
stockValue = message.readDouble();
stockTime = message.readLong();
stockDiff = message.readDouble();
stockInfo = message.readString();
```

- « *ObjectMessage* » : Ce type permet de transférer un objet java.

```
ObjectMessage message = session.createObjectMessage();
message.setObject(myObject);
```

La structure du message JMS

Le message manipulé par le MOM JMS est composé des parties suivantes:

- Une *entête*, qui a la même structure pour tous les messages, et contient principalement les champs nécessaires à l'identification et au routage du message.

- Des *propriétés*, qui viennent en quelque sorte *compléter l'entête*, avec des attributs spécifiques, soit définis par le MOM en complément de l'entête minimale JMS, soit définis par l'application pour ses besoins particuliers.

- Le *corps* du message, qui peut avoir différents formats: texte, objets Java ou données XML.

Les principaux champs de l'entête sont:

- *JMSMessageID* : identifiant unique du message

- *JMSDestination* : identification de la *queue* ou du *topic* destinataire du message

- *JMSCorrelationID* : utilisé pour synchroniser de façon applicative deux messages de la forme requête/réponse. Dans ce cas, dans le message réponse, ce champ contient le messageID du message requête

Selon l'image habituelle, l'entête correspond à ce qui est écrit *sur l'enveloppe*, le corps correspond à ce qui est *dans l'enveloppe*. Le MOM ne lit et n'utilise que les données de l'entête, y compris les *propriétés*. Ainsi, la *sélection de messages*, que l'on verra plus loin, peut dépendre de ces *propriétés*, mais non du corps du message.

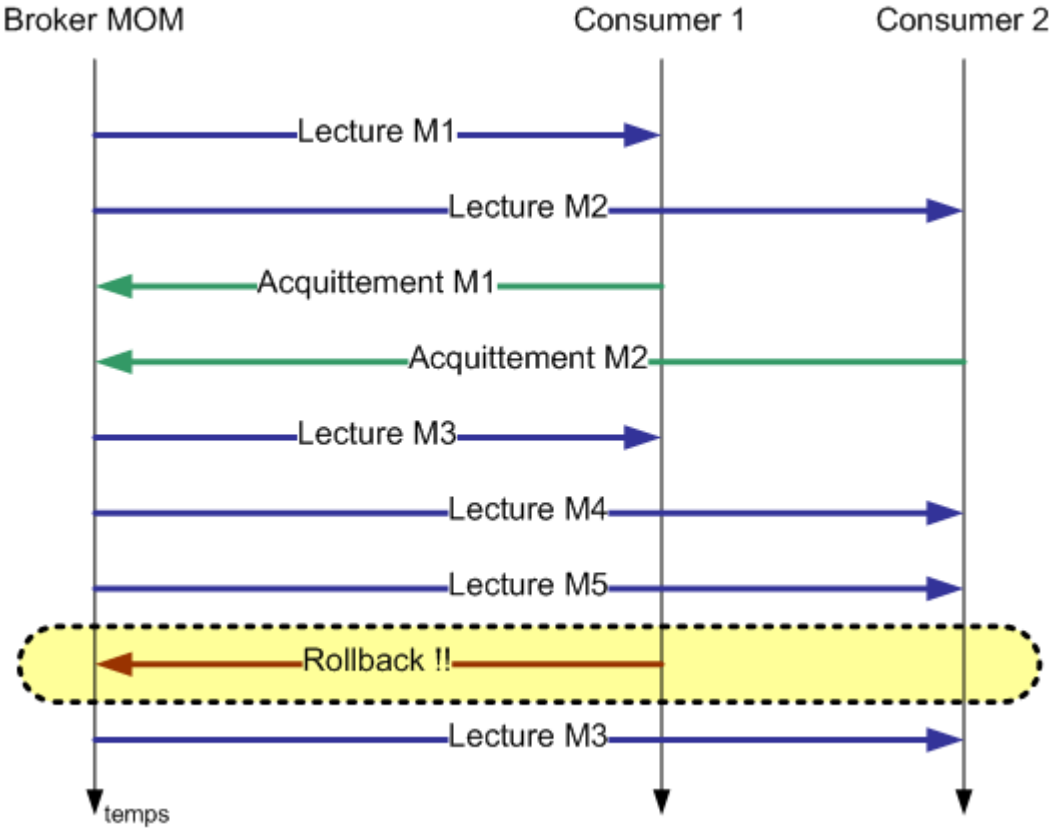
Ordre des messages

Le MOM garantit qu'un message sur une *queue* sera remis *au plus une fois*, mais il ne garantit pas que les messages seront remis dans l'ordre dans lequel ils ont été émis.

En fait, il y a presque une impossibilité théorique à garantir l'une et l'autre de ces deux propriétés: la remise unique, et la remise ordonnée. En effet, un consommateur peut lire un message, et ne l'acquitter que longtemps après. Si le consommateur n'acquitte pas, le message doit être recyclé. Ainsi pour assurer la remise ordonnée, le MOM devrait attendre que tous les messages jusqu'à N aient été non seulement reçus, mais acquittés, avant de livrer un message N+1, ce qui aurait un effet catastrophique sur les performances.

Nous verrons plus loin que les MOMs permettent une gestion des transactions, qui permet en quelque sorte d'annuler des opérations qui n'ont pas encore été validées, *commitées*, en ordonnant un retour arrière, un *rollback*. Voir « Gestion des transactions », page 29.

La figure suivante montre comment un rollback, soit explicite, soit implicite, c'est-à-dire provoqué par la fermeture de session, oblige à recycler un message alors que les suivants ont déjà été délivrés.



Durée de vie d'un message

L'application émettrice peut spécifier la durée de vie du message. Le message est donc 'valable' jusqu'à l'expiration de cette durée, au-delà le MOM peut le détruire sans l'avoir remis. La plupart des MOMs choisissent plutôt de l'aiguiller vers la *Dead Message Queue*, qui permettra de garder la trace de l'événement, et de recycler le message le cas échéant.

À noter que si l'on est dans le contexte d'une transaction, la durée de vie démarre quand même à l'instant d'émission, et non à l'instant du *commit*.

Priorité

Une fonctionnalité optionnelle, mais utile, proposée par le JMS, est la gestion des priorités, c'est-à-dire que la délivrance des messages s'effectue *selon leur priorité*.

Un message de plus haute priorité peut donc « doubler » un message de moindre priorité, pour autant que celui-ci n'ait pas encore été lu.

Remarquons que JMS 1.1 n'oblige pas les fournisseurs à implémenter cette fonctionnalité.

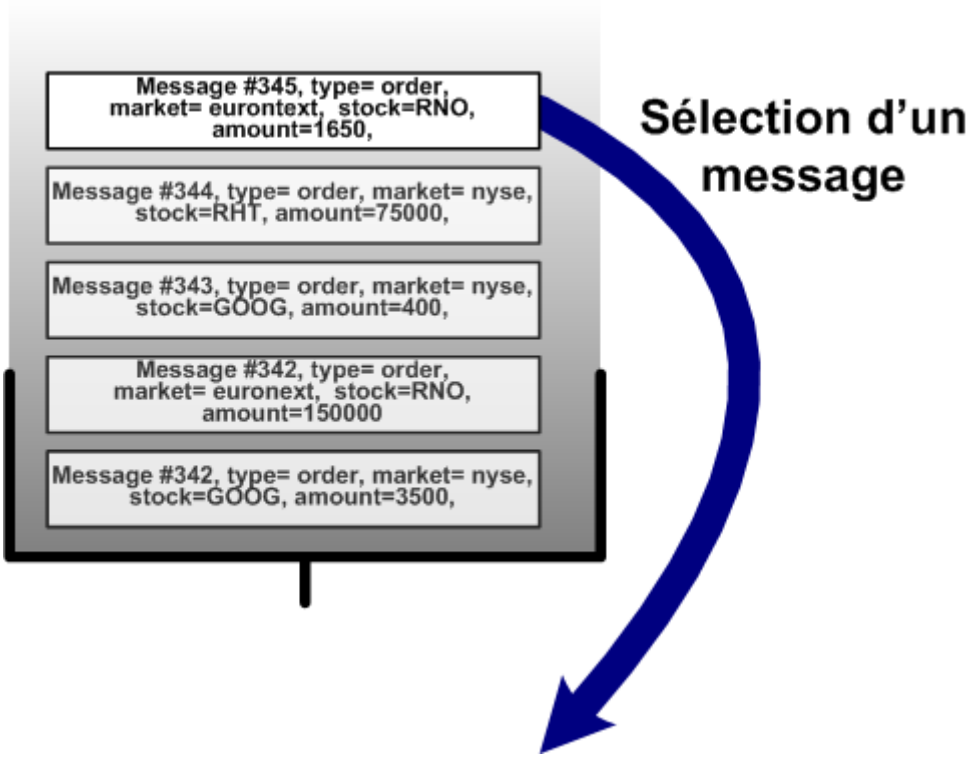
Sélection des messages

JMS prévoit que les applications clientes ont la possibilité de sélectionner les messages qu'elles lisent, sur la base des champs d'entête et de propriétés. On voit bien sûr que, s'il y a sélection, les messages ne seront forcément pas délivrés dans l'ordre.

La sélection des messages est définie dans JMS 1.1, elle est donc offerte par tous les MOMs étudiés. La syntaxe est inspirée du SQL, elle peut faire intervenir différents opérateurs de comparaison, d'expressions logiques, et même des opérations arithmétiques.

À titre d'exemple, imaginons une application qui communique avec une queue et lui envoie des messages avec les propriétés suivantes : *JMSType, market et amount*. Une application cliente ne souhaitant obtenir que les opérations sur le marché Euronext dont le montant est inférieur à 1 000 000 €, appliquera le *selector* suivant : *JMSType = 'order' AND market = "Euronext' AND amount < 1000000.*

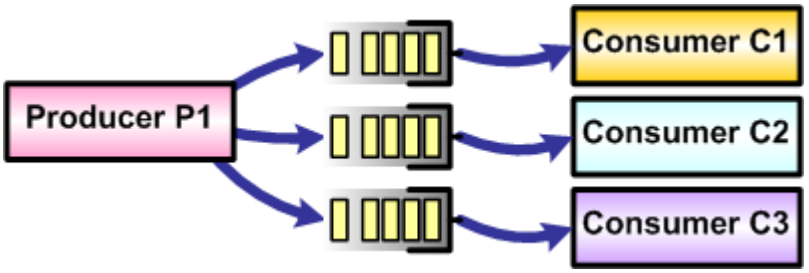
Certains MOMs peuvent accepter d'autres types de syntaxe, qui ne sont pas requis par JMS 1.1, typiquement Xpath. Mais dans tous les cas, la sélection porte sur entête et propriétés, et non sur le corps du message.



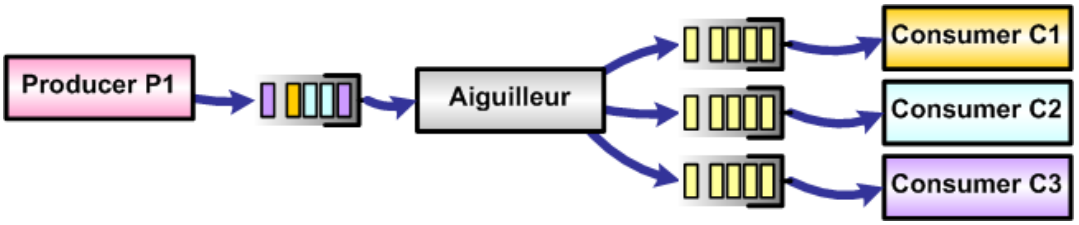
Aiguillage et spécialisation

On peut donc mettre en œuvre, au moyen de la sélection, une *spécialisation* des consommateurs. En fait, dans une logique d'affectation et de répartition de tâches, on peut distinguer trois techniques:

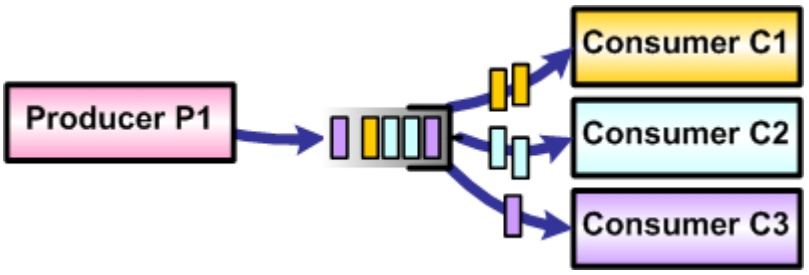
- L'application émettrice, *producer*, place des messages dans des queues différentes selon la nature de la tâche à effectuer. Et une application spécifique est en lecture sur chacune des queues.



- L'application émettrice place les messages dans une queue unique, mais la queue est ensuite éclatée en plusieurs queues, ceci soit au moyen d'une application relais jouant un rôle d'aiguillage, soit au moyen d'un *traitement d'aiguillage*, si le MOM le permet.



- L'application émettrice place les messages dans une même queue, et les applications *consumer* sélectionnent les messages selon leur spécialisation. Ici « Consumer C1 » prend les messages jaunes, C2 les messages bleus, C3 les messages violets. L'application *producer* n'a pas à connaître cette répartition.



Trois manières de gérer à peu près le même problème, à différents niveaux. Dans le premier cas la logique d'aiguillage est intégrée au *producer*, dans le dernier cas, elle relève du *consumer*, et dans le cas intermédiaire, elle est déportée dans une application dédiée.

Synthèse JMS

JMS est une API, et cette API correspond à des *services d'échange* entre des producteurs et des consommateurs de messages, s'appuyant sur des concepts que nous avons présentés. Au-delà de l'API donc, JMS définit *les fonctionnalités* centrales des MOMs.

JMS spécifie le service, mais ne spécifie pas *comment* ce service est mis en œuvre. Chaque fournisseur, *JMS Provider*, est libre de ses choix d'implémentation.

Comme on l'a vu plus haut, les protocoles d'échanges peuvent également être considérés comme des choix d'implémentation propres à certains MOMs, même si nous considérons qu'ils ont une réelle importance.

La spécification JMS n'est pas en tous points complète. Certaines fonctions essentielles au fonctionnement d'une plateforme MOM ne sont pas décrites dans la spécification et font donc l'objet d'implémentations particulières. C'est le cas en particulier pour la configuration et l'administration du service de messagerie, pour la sécurité (intégrité et

confidentialité des messages) et pour certains paramètres de qualité de service.

Par ailleurs, la plupart des MOMs proposent des fonctions additionnelles qui se présentent comme des atouts spécifiques par rapport aux offres concurrentes (par exemple les *topics* hiérarchisés, des fonctions de sécurité et des mécanismes de haute disponibilité, etc.). Bien sûr, la mise en œuvre de ces fonctionnalités se fait au détriment de la capacité à changer de MOM, en respectant l'API JMS.

Comme d'autres spécifications d'interface, comme le SQL par exemple, la promesse de pouvoir changer d'implémentation de MOM JMS de manière transparente, n'est pas facilement tenue. Mais ce n'est pas très grave. La spécification commune apporte déjà le bénéfice d'une communauté de vision, d'approches, et de compétences. Un architecte peut raisonner sur la base d'un MOM sans savoir nécessairement de quelle « marque » il sera, et un développeur qui a pratiqué JMS avec un premier MOM, pourra presque immédiatement en pratiquer un second.

Caractéristiques principales des MOM

Nous parcourons ici les principales classes de fonctionnalités offertes par les MOMs, en identifiant les possibilités communes à tous les outils, et celles qui sont plus spécifiques.

Langages d'implémentation, APIs et environnements supportés.

Les MOMs open source que nous étudions ici sont tous codés en Java. Nous ne les avons pas sélectionnés sur ce critère, mais il se trouve que tous les éditeurs concernés ont fait ce choix. Il est assez naturel puisque le MOM doit souvent s'insérer dans un environnement hétérogène, en termes de systèmes d'exploitation et de serveurs. La portabilité est donc primordiale, et elle est l'un des atouts majeurs de l'environnement Java. S'ajoute à cela, la disponibilité dans cet environnement de librairies puissantes et éprouvées, pour les fonctionnalités fondamentales en matière de réseau, de sécurité, d'accès à des bases de données, de gestion transactionnelle, etc.

Cela dit, le langage dans lequel le MOM lui-même est codé pourrait être d'une importance secondaire. De même qu'il importe peu de savoir dans quel langage MySql est codé, du moment que nous pouvons en invoquer les fonctionnalités depuis divers environnements. Ce qui importe pour les applications, c'est la disponibilité d'APIs, de fonctions ou méthodes qui peuvent être appelées pour invoquer les services du MOM.

Mais certains MOMs se sont largement focalisés sur l'environnement Java, y compris pour les APIs, c'est-à-dire qu'ils n'offrent pas d'APIs pour d'autres environnements. C'est, selon nous, un handicap majeur, car la

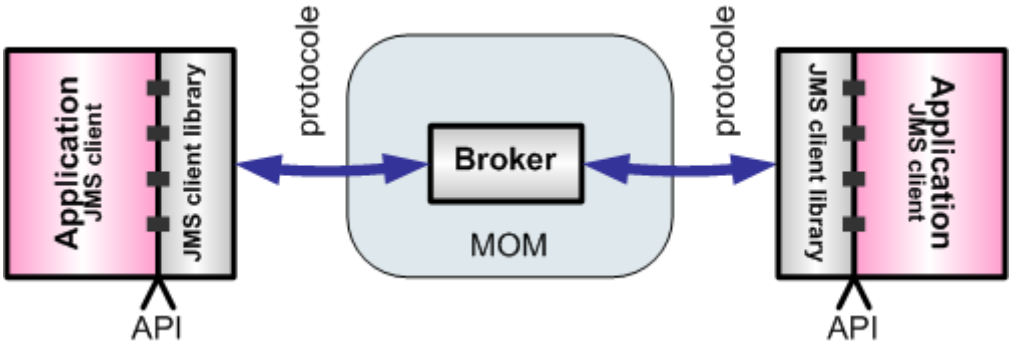
capacité à relier des applications diverses, à gérer l'*hétérogénéité*, est précisément une des finalités du MOM. S'il ne peut être mis en œuvre qu'entre des applications Java, il perd une partie de son utilité.

Lorsque le MOM offre des APIs pour d'autres environnements que Java, elles se présentent sous la forme de librairies de fonctions dans l'environnement cible, par exemple en C ou en PHP.

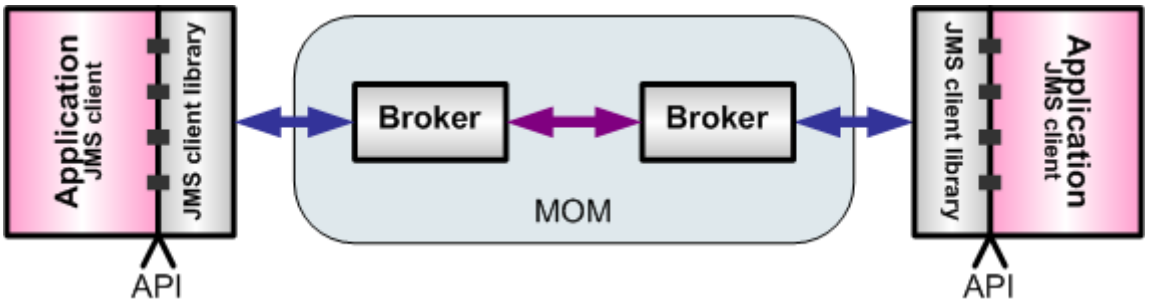
La figure suivante permet de bien distinguer ces notions:

- L'API proprement dite, qui est l'interface appelée par l'application.
- Les librairies du provider, invoquées par cet API, représentées ci-dessous en tant que « JMS Provider API »
- Le *Broker*, qui est un processus indépendant de l'application, en charge de la gestion des messages.

Les fonctions de la librairie JMS échangent avec le *broker* par un protocole réseau.

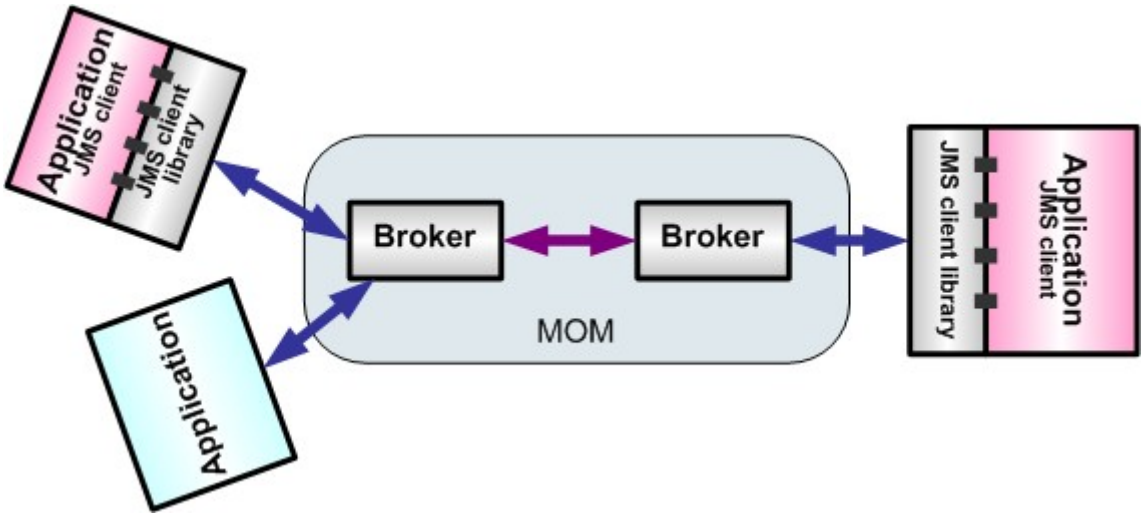


L'échange peut impliquer plusieurs *brokers,* qui échangent entre eux. Le protocole *interne* du MOM, entre *brokers*, peut être le même, ou bien différer du protocole *externe*.



Rappelons que, par définition, JMS est une API *pour l'environnement Java*. Dans les exemples précédents, les applications sont donc nécessairement Java.

Si le protocole d'échange avec le broker est standard, une application peut, théoriquement, échanger directement avec le broker, sans passer par une librairie de fonctions. Il suffit qu'elle respecte le protocole d'échange avec le broker. Mais mettre en œuvre un protocole réseau est assez complexe, et source d'erreurs, de sorte que ce n'est pas le rôle d'une application en général.



Sur l'exemple ci-dessus, on est en environnement hétérogène : certaines applications invoquent le MOM via les APIs fournies, tandis que l'application bleue échange directement avec le broker selon le protocole réseau.

Des APIs peuvent être fournies pour d'autres environnements que JEE, par exemple C++, PHP, .Net, Ruby, Perl. Plus la liste de langages grâce auxquels on peut accéder au MOM est grande, meilleures sont les possibilités d'intégration.

Protocoles

Lorsqu'une application appelle une API pour invoquer le MOM, la fonction d'API prend en charge l'échange avec un *broker* du MOM.

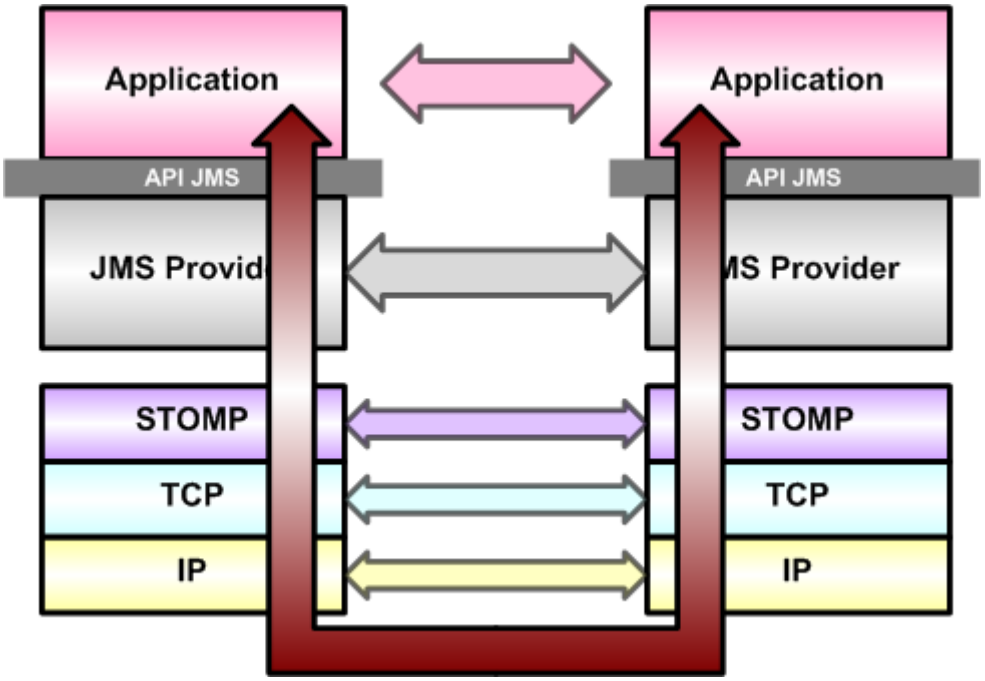
L'échange entre l'application et le broker implique un *protocole*. Le protocole définit comment les services du MOM seront spécifiés, leurs paramètres, et le format des messages. Par exemple, le protocole doit spécifier que le nom d'une queue de message est représenté par une chaîne de caractères codés en UTF8.

On peut distinguer des protocoles *externes*, entre application et brokers, et des protocoles *internes*, entre brokers.

Il existe deux standards en matière de protocole MOM : AMQP (Advanced Message Queuing Protocol) et STOMP. On les appelle des « *wire-level*

protocols » (protocoles filaires), dans le sens où ils sont en charge de gérer les échanges sous la forme d'une suite d'octets transmis.

Comme toujours en matière de communication réseau, on a affaire à une *pile* de protocoles, c'est-à-dire que le protocole du MOM s'appuie lui-même sur des couches de protocoles inférieures. Ainsi, STOMP peut s'appuyer, à la manière du HTTP, sur une pile TCP/IP. On appelle *support de communication logique* le protocole de transmission du message, par exemple STOMP dans l'exemple précédent.



Le schéma précédent fait apparaître un exemple de pile de protocoles. Les différentes flèches horizontales représentent les échanges virtuels, aux différents niveaux : au niveau le plus haut, une application échange avec une autre, en fait le broker d'un *JMS provider* échange avec son homologue. Les messages descendent puis remontent la pile des protocoles, comme classiquement.

Notons que du côté des MOMs, on parle souvent de « *connecteurs* » pour parler des différents protocoles.

Traitement des messages par le MOM

La fonction naturelle, essentielle, d'un MOM n'est pas d'effectuer des traitements sur les messages qui lui sont confiés. Sa fonction est de les acheminer de manière fiable jusqu'à leur destinataire. C'est même ce qui distingue le MOM d'un EAI ou bien d'un ESB: il achemine les messages et c'est tout. En particulier, le MOM ne « regarde » pas le contenu des messages, ce n'est pas son problème.

Pourtant, l'un des MOMs que nous étudierons, ActiveMQ, offre cette possibilité supplémentaire, de définir des traitements à exécuter sur les messages qui lui sont confiés. Ces traitements sont définis en référence aux différents *Enterprise Integration Patterns,* un recensement des familles de traitements (cf « Enterprise Integration Patterns », page 35.

Un cas simple, par exemple, est un traitement d'aiguillage, en fonction du contenu du message: le message concerne sur des ordres de bourse, si l'ordre porte sur une valeur EuroNext, il doit être routé sur une queue A, s'il porte sur une valeur du NYSE, il doit être routé sur une queue B.

Un autre exemple serait une règle d'envoi d'une copie: si le montant de l'ordre de bourse est supérieur à 1 million, alors il faut envoyer un message en copie sur une queue C.

La question importante est: Est-ce une bonne idée d'insérer ces règles et ces traitements dans le MOM ? Ne sont-ils pas plutôt du ressort de l'application ? Le MOM ne devrait-il pas plutôt rester dans son rôle de tuyauterie passive ?

La réponse n'est pas immédiate. Sortir certaines règles des applications peut être un moyen de gagner en flexibilité, d'intervenir dans la gestion des flux sans modifier les applications. Mais si l'on met en œuvre de tels traitements de manière massive, alors on a en fait *éparpillé des morceaux d'applications dans le middleware*, et cela au détriment de la maintenabilité, et de la cohérence de vision.

Quoi qu'il en soit, si le MOM n'offre pas de telles possibilités, ou bien qu'on ne veut pas en faire usage, il est toujours possible, et même aisé, de les mettre en place dans des applications relais.

Gestion des transactions

On peut distinguer trois niveaux dans la gestion transactionnelle des messages:

- La gestion des acquittements
- La gestion des transactions JMS
- La gestion des transactions XA

Gestion des acquittements

L'application destinataire, qui *consomme* les messages, doit généralement effectuer un *traitement* qui dépend de ce message. Le message ne doit donc pas seulement être *lu*, il doit être *traité*. C'est une distinction importante, dans la mesure où l'application pourrait s'arrêter brutalement (bug ou bien panne matérielle) entre l'instant où elle a lu le message et l'instant où elle a fini de le traiter avec succès.

C'est pourquoi le fonctionnement normal de l'application *consommatrice* est en trois étapes:

1. Recevoir un message

2. Traiter le message

3. Acquitter le message, c'est-à-dire notifier la bonne fin du traitement.

Tant que le message n'a pas été acquitté, il est conservé par le broker. Si le message n'est jamais acquitté, il est recyclé, c'est-à-dire qu'il sera remis lors d'un prochain appel d'une application cliente. Notons que c'est ce principe qui rend presque impossible la garantie de délivrance ordonnée pour les MOMs en général.

Une application cliente peut acquitter en un seul appel, tous les messages reçus et encore non acquittés. C'est donc une forme de gestion transactionnelle en lecture.

Transactions JMS

Il est possible de réunir différents ordres d'émission et de réception de messages en une transaction, un ensemble *insécable* d'opérations. C'est-à-dire que soit toutes ces opérations seront exécutées avec succès, soit aucune d'entre elles ne sera exécutée.

Comme pour les bases de données, l'application ouvre une transaction, effectue différentes opérations JMS, puis termine la transaction par un ordre *commit*. Si l'application détecte une condition d'erreur qui interdit de terminer avec succès l'ensemble des opérations, elle demande un *rollback*, c'est-à-dire un retour arrière sur toutes les opérations précédentes. Si l'application « se plante », et donc s'interrompt sans avoir fait ni *commit*, ni *rollback*, un *rollback* sera exécuté de manière implicite. Dans le cas d'émissions de messages, aucun message n'a en fait été émis avant le *commit*. Dans le cas de réception de messages, aucun acquittement n'aura été exécuté avant le *commit*.

Il y de nombreux usages de ces transactions JMS, d'une manière générale pour assurer la *cohérence:*

- Une application peut par exemple émettre 10 messages et être assurée que soit tous seront bien émis, soit aucun ne le sera.

- Une application qui jouerait un rôle de relais pourra ainsi lire un message sur une queue, le traiter, et écrire un message résultant sur une queue en aval, tout cela au sein d'une transaction, et donc avec la garantie de ne pas perdre de message si elle est interrompue entre la lecture et l'écriture.

- Enfin, de la même manière, une application qui doit réceptionner plusieurs messages avant d'effectuer un traitement, peut réunir ces lectures en une même transaction.

Voici un petit exemple de code Java utilisant les transactions.

```
session = connexion.createTopicSession(true, Session.AUTO_ACKNOWLEDGE);

void onMessage(Message msg){
  try{
        // un traitement, susceptible de lever une exception
        m2 = …;
        publisher.publish(m2);
        session.commit(); // acquittement des messages
    }catch(Exception e){
        session.rollback(); // annulation des messages
  }
}
```

Si le traitement réussit, le programme client exécute un *commit*, sinon, il demande un *rollback*, c'est-à-dire qu'il ordonne au broker tout annuler.

Transactions XA

Enfin, la troisième manière de gérer les transactions s'inscrit dans le cadre de « XA » en environnement Java. XA est une spécification définissant les interfaces qui permettent de mettre en œuvre des transactions hétérogènes, c'est à dire s'étendant à plusieurs *ressources* de différentes natures, telles que bases de données, serveurs d'application (EJB), ainsi donc que *les MOMs*. En environnement Java, XA est disponible via l'API JTA, Java Transaction API.

Il s'agit donc de réunir, dans un même ensemble insécable, indivisible, des traitements portant sur ces diverses ressources.

Un cas très simple et typique est celui d'une application qui:

- Lit un message auprès d'un broker de MOM
- Effectue une écriture sur la base de données.

En l'absence de transactions XA, l'application devrait acquitter son message auprès du MOM soit avant, soit après, l'écriture en base. Mais si elle acquitte avant, puis se plante, elle n'a pas effectué l'écriture, mais le message est pourtant considéré traité avec succès. Si à l'inverse elle écrit dans la base en premier, mais se plante avant d'avoir acquitté le message, alors le message sera recyclé, et il y aura donc eu deux écritures.

Sur cet exemple très simple, on voit donc que les transactions XA peuvent être absolument indispensables dans certains contextes afin d'assurer une réelle garantie de cohérence au niveau global du système d'information. Bien entendu, les transactions peuvent être sensiblement plus larges et plus complexes.

Dead Message Queue

Même si ce n'est pas requis par la spécification JMS, différents MOMs définissent une queue spéciale appelée *« Dead Message Queue »* ou DMQ, qui correspond à une sorte de poubelle, où l'on pourra retrouver des messages qui auraient pu être perdus pour différentes raisons techniques.

Généralement, la DMQ reçoit les messages :

- qui n'ont pas une destination valide.
- dont la destination est remplie. (limite, plus de mémoire, ...)
- dont la durée de vie (TTL, *time to live*) a expiré
- qui se sont fait rejeter un certain nombre de fois (configurable). Ces messages apparaissent comme des messages « poisons » polluant la plateforme. Par exemple, un message qui fait planter systématiquement un client est un message polluant.

Bien sûr, il convient qu'un administrateur analyse ces messages pour en déterminer les causes d'erreur éventuelles. La DMQ contribue à garantir qu'aucun message n'est perdu par le MOM, il est donc naturellement recommandé qu'elle soit persistante.

Persistance des messages

Comme on l'a vu en introduction, *la fiabilité et la robustesse* sont deux qualités essentielles, constitutives des MOMs, c'est-à-dire qu'un MOM doit acheminer un message qui lui a été confié, sans jamais le perdre, même en présence d'événements inattendus.

Si l'application destinatrice n'est pas en mesure de recevoir le message, le MOM peut être amené à le conserver un temps indéfini. Or le MOM lui-même peut être arrêté, que ce soit du fait de pannes matérielles ou pour des raisons de maintenance.

Pour garantir que les messages ne seront pas perdus, le MOM doit donc les stocker de manière sécurisée, de manière persistante.

Il est possible de faire fonctionner un MOM dans un mode sans persistance, c'est-à-dire dans un mode où les messages sont seulement conservés en mémoire. On peut choisir ce mode pour atteindre des performances plus élevées – car *la persistance a un coût* – au détriment bien sûr de la fiabilité.

La persistance est toutefois importante, voire essentielle, dans les cas suivants :

- Lorsque les messages sont critiques, par exemple s'il s'agit de transactions financières.
- S'il peut y avoir un *déséquilibre positif* entre producteurs et consommateurs, c'est-à-dire que de manière durable les applications productrices émettent plus de messages que les applications destinatrices ne peuvent lire et traiter. Les capacités mémoires risqueraient d'être dépassées.
- Lorsque le traitement des messages est fortement asynchrone, de manière structurelle, c'est-à-dire par exemple si les messages ne sont traités qu'en fin de journée, d'une manière que l'on pourrait assimiler à un traitement batch.
- Lorsque l'on doit mettre en œuvre une gestion des transactions, qui implique une utilisation plus importante de la mémoire. Les messages utilisant les transactions ne sont supprimés que lorsque les transactions sont validées.
- En présence de réplication, lorsqu'elle est offerte. Il est nécessaire de mettre en œuvre la persistance pour accroître les possibilités de stockage : un broker ne pourra gérer la réplication de tous les domaines de la plateforme MOM en mémoire.

La persistance peut être mise en œuvre par le MOM de différentes manières :

- Sur de simples *fichiers*
- Sur une *base de données* relationnelle
- Au moyen d'un dispositif spécifique combinant deux supports de persistance.

Le stockage sécurisé des données et leur gestion transactionnelle étant un problème déjà parfaitement résolu par les SGBD relationnels, la plupart des MOMs appuient leur persistance sur une telle base. L'accès par JDBC leur permet de supporter un large éventail de gestionnaire de base de données (Mysql, Postgres, Oracle, DB2, ...), y compris des bases 100% java telles que Hypersonic et Derby.

Tous les MOMs étudiés ici supportent la persistance via JDBC. Cependant, chaque MOM stocke différemment les données. Certains introduisent un mode de persistance optimisé. Ils sont amenés parfois à combiner trois types de stockage : fichier, base de données et mémoire. Et ceci dans le but d'optimiser la *fiabilité et la performance*. C'est donc un

aspect que nous développerons pour chacun des outils, et le dernier chapitre présentera les résultats de différents tests de performance.

Fonctionnalités avancées

Code générique et JNDI

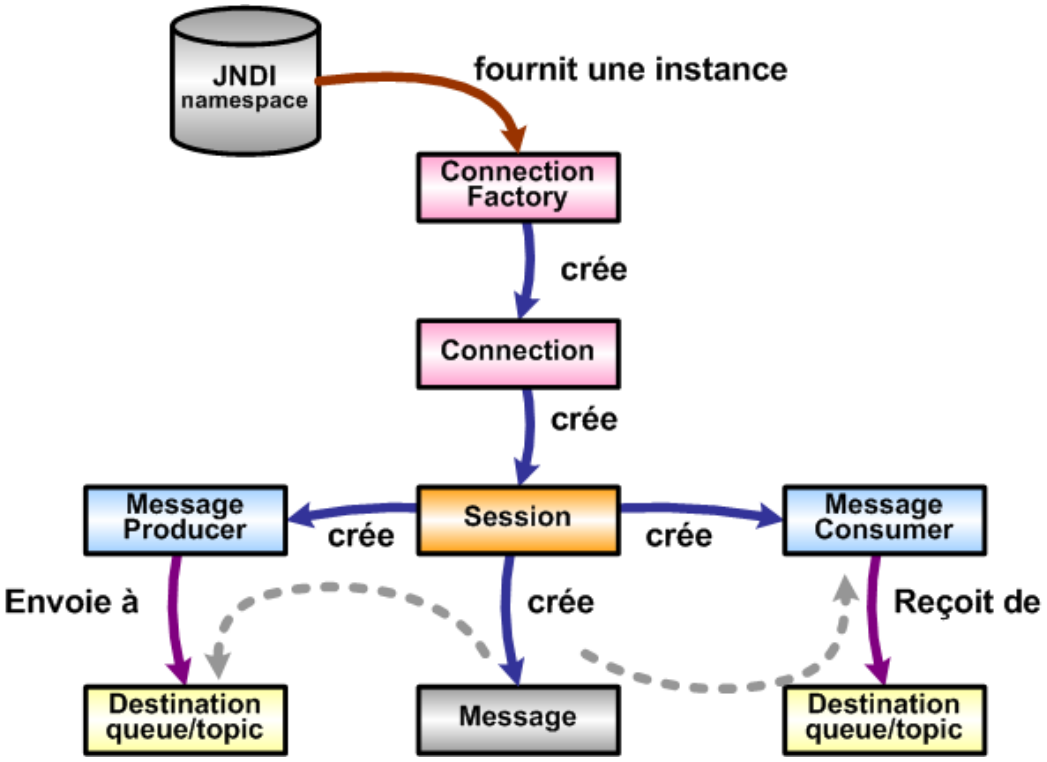
Comme nous l'avons souligné, le principe d'une spécification telle que JMS est que l'on peut écrire un programme s'interfaçant à un *provider JMS*, c'est-à-dire à un MOM, sans être dépendant d'une implémentation particulière, d'un MOM en particulier.

Pour cela, le programme *« JMS Client »*, ne doit pas instancier directement les classes du MOM, et la bonne pratique est de les obtenir à partir d'un fournisseur JNDI.

De même que JDBC est une interface permettant d'accéder à une base de données, de même JNDI ou *« Java Naming and Directory Interface »* est l'interface qui permet l'accès à des services de nommage et de répertoire de façon standard. L'utilisation la plus commune de l'interface JNDI concerne l'accès à un annuaire LDAP. Mais au-delà de la fonctionnalité usuelle de gestion d'une base de personnes, d'utilisateurs, on peut utiliser l'API JNDI simplement pour accéder à des *objets* désignés par des *noms*. Ainsi, dans le contexte des MOMs, JNDI sert à stocker des objets génériques du MOM, afin de transmettre leur implémentation spécifique de JMS au programme.

Le premier objet que le programme obtient est une *connectionFactory*, une usine à connexions. Puis la *connectionFactory* permettra de créer un objet *Connection*, à partir duquel on créera un objet *Session*, qui lui-même pourra instancier des objets *Message*, *MessageProducer* et *MessageConsumer*.

Ce que l'on peut représenter comme suit :



Enterprise Integration Patterns

Le livre de Gregor Hohpe et Bobby Woolf intitulé « *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions* », est un ouvrage de référence en matière de middleware. Il recense en particulier toutes les formes d'interactions par middleware, et tous les types de traitements que peut réaliser un middleware. Par exemple un traitement de routage d'une queue vers une autre, selon différentes règles. Ou encore des traitements de fusion ou de fission des messages : le moteur de traitement peut éclater un message en plusieurs, ou à l'inverse réunir différents messages en un seul.

EIP se ne limite pas à cela. Il décrit toutes les manières à disposition pour intégrer des logiciels entre eux. Catégorisant ces patterns selon leur objet, c'est un peu la bible des architectes et urbanistes.

Voici la liste des catégories référencées par EIP ainsi que quelques exemples :

- *Styles d'intégration* : Liste les supports de communications comme le transfert de fichier, le partage de donnée, l'invocation de procédure et la communication par message. On retrouve ici les type de middleware évoqués en introduction

- *Les systèmes de messagerie* : Cette catégorie de *pattern* regroupe les composants des systèmes basés sur une communication par messages comme un message, un traducteur de message, un routeur de message, …

- *Les cas d'utilisation d'une communication par messages* : Cette catégorie décrit le concept de queue, de topic, de bridge de message et autres

- *Méthode de construction des messages* : message de commande, message de document, message d'événement, …

- *Le routage des messages* : routage basé sur le contenu, agrégation de message, …

- *Transformation de message* : envelopper un message, enrichir le contenu, …

- *Réception de message* : Cette catégorie décrit les différentes manières de recevoir un message comme la consommation à callback, abonnement durable, sélection de message, …

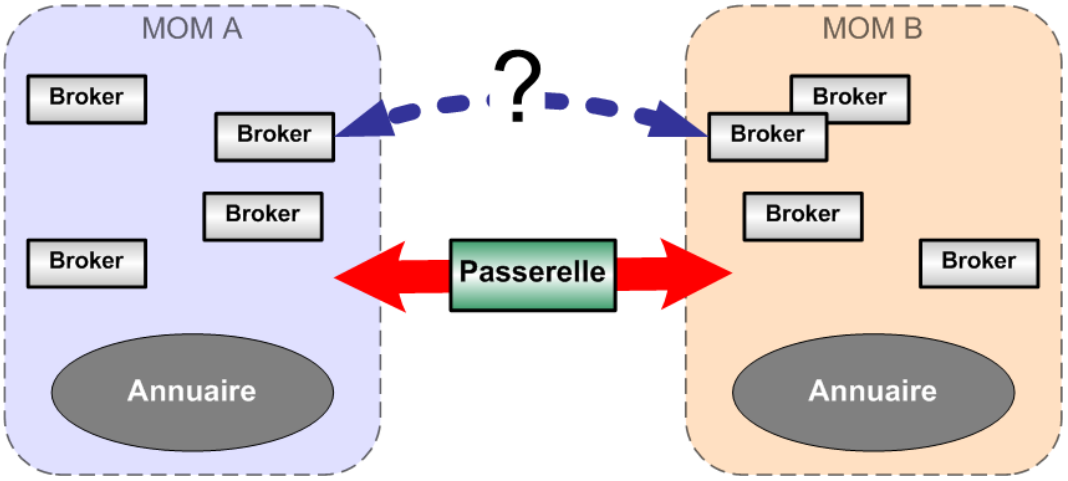
- *Administration de la plateforme* : Cette catégorie décrit les différentes manières de gérer la plateforme : persistance, détournement de message, écoute passive, …

Interopérabilité entre MOMs

Les *protocoles filaires* des MOMs (par exemple entre une application et un broker ou entre un broker et un autre) sont parfois sans spécification et sans documentation. Parmi les MOMs de notre sélection, aucun n'offre nativement une passerelle vers d'autres MOMs.

Pour résumer, la partie haute de la figure ci-dessous, c'est-à-dire l'interconnexion des MOMs au niveau du protocole interne, n'est pas possible. Il faudrait que les deux MOMs utilisent le même protocole interne, ce qui n'est en général pas le cas.

Il ne suffit pas d'assurer la transmission des messages, il faut gérer la propagation de tout l'annuaire des domaines.



À des fins d'interopérabilité, certains MOMs ont mis en place un système dit de « *Bridge* » (Passerelle). C'est une application à deux faces qui est, d'un côté connectée à un MOM et de l'autre connectée à un autre. Lorsqu'elle reçoit un message d'un côté, elle le transmet de l'autre.

Cette solution peut rencontrer des limites en termes de performance, flexibilité et de sécurité. Hormis le temps et la complexité de mise en place, la passerelle risque d'être un *goulot d'étranglement*, et un point de fragilité. Au sein d'un même système d'information, on vise à l'évidence, un MOM unique. Mais bien sûr, les cas possibles d'hétérogénéité sont nombreux : rachat et intégration d'entreprise, relations avec des partenaires, etc.

Passerelle à base d'ESB

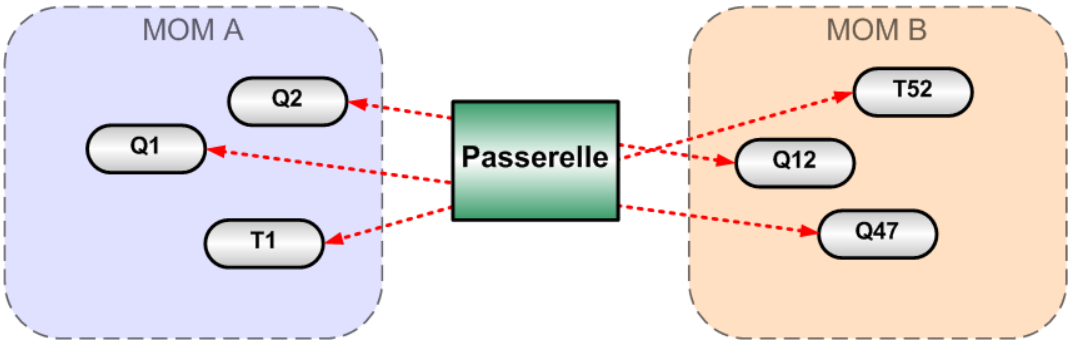
Le travail d'intégration est laissé aux solutions du type *EAI ou ESB* (Enterprise Service Bus). À l'aide de l'ESB Mule, par exemple, il est assez simple de mettre en place une passerelle entre deux domaines de deux MOMs - Pas besoin d'application supplémentaire pour jouer le rôle de passerelle ni même de toucher à une ligne de code Java. Regardons comment configurer Mule pour cette tache. Pour ce faire, il faut créer deux connecteurs : un vers chacun des MOMs. Puis il faut créer un service par domaine qui aura la mission de transmettre les messages. Voici une partie du fichier de configuration.

```
[...]
<jms:connector name="jmsConnectorJBOSS"
     connectionFactoryJndiName="java:/ConnectionFactory"
     jndiInitialFactory="org.jnp.interfaces.NamingContextFactory"
     jndiProviderUrl="jnp://localhost:1099"
     jndiDestinations="true"
     forceJndiDestinations="true"
     specification="1.1"/>
<jms:connector name="jmsConnectorWEBLOGIC"
     jndiProviderUrl="t3://localhost:7001"
         connectionFactoryJndiName="javax.jms.QueueConnectionFactory"
         jndiDestinations="true"
```

```
         forceJndiDestinations="true"
         jndiInitialFactory="weblogic.jndi.WLInitialContextFactory"
         specification="1.0.2b"/>

<model name="JMSBridge">
<service name="JBOSS_WebLOGIC">
        <inbound>
          <jms:inbound-endpoint topic="my.destination" connector-
ref="jmsConnectorJBOSS"/>
        </inbound>
        <outbound>
          <pass-through-router>
          <jms:inbound-endpoint topic="my.destination" connector-
ref="jmsConnectorWEBLOGIC"/>
          </pass-through-router>
        </outbound>
</service>
</model>
```

La tâche n'est pas d'une grande complexité, mais elle peut être fastidieuse, et donc coûteuse, puisqu'il faut relier des domaines entre eux un par un, sans en oublier aucun. Et bien sûr, cette configuration devra être l'objet d'une maintenance, en fonction des variations de configuration intervenant de part et d'autre.



Dans notre exemple, la passerelle met en correspondance :

- La *queue* Q2 du MOM A à la *queue* Q47 du MOM B
- La *queue* Q1 du MOM A à la *queue* Q12 du MOM B
- Le *topic* T1 du MOM A à la *queue* T52 du MOM B

Dans la pratique, cette passerelle est généralement réalisée en Java et utilise le JMS. On parle de « *JMS Bridge* » ou de *passerelle JMS*.

La mise en place d'une passerelle rend caduque certaines fonctionnalités incluant plusieurs brokers. Des fonctionnalités comme le partage de média de stockage ou le clustering ne marcheront plus de manière naturelle. Deux MOMs différents impliquent deux politiques différentes de persistance, de réplication, de topologie.

Ceci étant, il est possible de *multiplier les passerelles* à des buts de répartition de charge ou de robustesse uniquement dans les cas de liaison de queue. D'autres solutions sont envisageables, mais cela reste des développements spécifiques relatifs à des problématiques d'intégration. Un exemple simple serait de bufférisер les transactions. Il est en effet bien *plus performant de regrouper la réception ou l'envoi de plusieurs messages dans une seule et même transaction*.

Gestion de la sécurité

Étant donné le rôle souvent central d'un MOM dans un système d'information, les questions de sécurité sont évidemment cruciales. Si n'importe quelle application peut se connecter au MOM et se mettre en lecture sur une *queue*, on voit qu'il sera facile de pirater le système et d'accéder à des données critiques, ou d'injecter des messages.

Un MOM interagit avec des applications, lesquelles interagissent avec d'autres applications, ou avec des utilisateurs. La question de la sécurité dans le contexte des MOMs est semblable à ce qu'elle est dans le contexte des bases de données. Les brokers doivent authentifier les applications qui s'y connectent, mais ils doivent aussi contrôler les droits spécifiques de chaque application vis-à-vis de chaque *opération* sur chaque *queue* ou *topic*. Et les brokers doivent aussi authentifier les autres brokers avec lesquels ils échangent.

Il est essentiel de mettre en place toute la politique de sécurisation du MOM dès son premier déploiement, quelle que soit la nature des informations échangées, ou la configuration réseau, car une fois le MOM institué comme standard d'échange, il est à craindre qu'on ne se reposera pas la question de la sécurité pour chaque nouvelle application qui en aura l'usage.

Les MOMs que nous étudions offrent la possibilité de spécifier les règles d'authentification et d'habilitations au moyen d'un *provider* de sécurité, utilisant le cadre de JAAS, *Java Authentication and Authorization Service.* Le MOM propose son propre *plugin* JAAS, dont le comportement est configuré par un fichier Xml, ce qui convient le plus souvent, mais il est envisageable également de mettre en place un plugin JAAS spécifique.

Administration et monitoring

Les MOMs offrent différentes possibilités d'administration et de monitoring :

API spécifique

Configuration et déploiement

Les MOMs peuvent fournir plusieurs modes de configuration : fichiers de configuration, messages adressés aux brokers, à travers différentes syntaxes (Ini, Spring, DSL, …), plus ou moins compliquées. On remarque une tendance à intégrer le MOM au sein d'environnements comme *Spring*. L'intérêt d'intégrer la configuration à Spring est par exemple la possibilité de lancer un broker à partir d'un outil le supportant. Ci-après un exemple issu de Mule.

```
<spring:beans>
      <spring:bean id="activeMqConnectionFactory1"
            class="org.apache.activemq.xbean.BrokerFactoryBean">
            <spring:property name="config"
                  value="file:conf/activemq/global/activemq_1.xml" />
            <spring:property name="start" value="true" />
      </spring:bean>
</spring:beans>
```

Dans certains cas, le MOM est intimement intégré à un serveur d'application - c'est le cas de JBoss - et ainsi utilise ses fichiers de configuration. Cette intégration est plutôt une gêne qu'autre chose.

Les MOMs peuvent aussi permettre de modifier leur configuration à chaud. Par exemple, il est utile d'avoir la possibilité d'ajouter des brokers à la volée sans avoir à redémarrer la plateforme, qui impliquerait une interruption. Les messages non persistants doivent être sauvegardés et remis en mémoire lors du démarrage, ce qui ne se fait pas automatiquement d'ailleurs.

Les MOMs étudiés sont tous réalisés en Java. Ils sont tous utilisables sur les plateformes supportant le Java 5 (Linux, Windows, Mac OS, Solaris, HP UX, AIX …).

Répartition de charge applicative

On parle parfois des *queues* comme mettant en œuvre un échange *« de 1 vers 1 »*. C'est exact *pour un message donné*, mais ce n'est pas nécessairement le cas pour *l'ensemble du flux de messages.* On a vu en effet que plusieurs applications clientes pouvaient être en lecture sur une même queue. Dans ce cas, le MOM délivre chaque message à une et une seule des applications. Les règles de choix de l'application ne sont

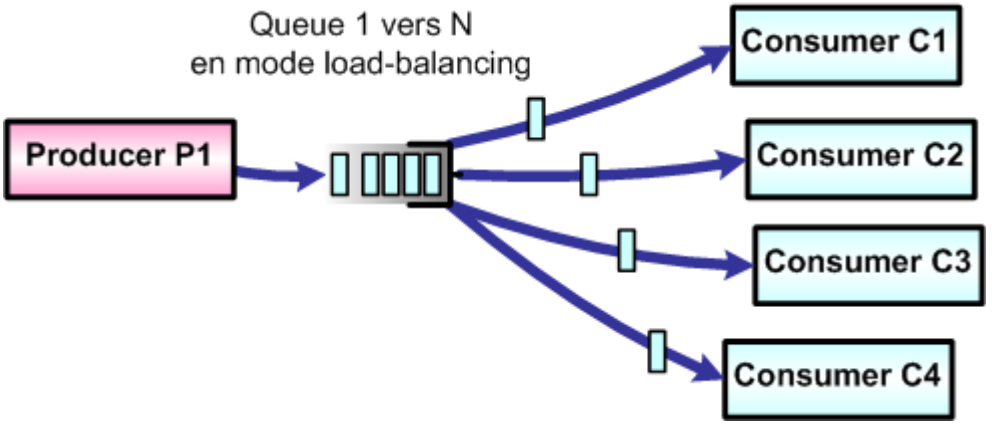
pas spécifiées, mais le plus souvent il s'agit d'un simple *round robin*, c'est-à-dire une attribution cyclique, « à tour de rôle ».

Ainsi, un MOM peut offrir un moyen très simple et robuste de mettre en œuvre une répartition de charge applicative.

Considérons que chaque message représente une *demande de traitement,* un par exemple un traitement d'OCR (reconnaissance de caractères) qui consomme beaucoup de CPU. Une application principale est en charge de définir chaque traitement unitaire, dont elle écrit les caractéristiques dans un message, qu'elle adresse sur une *queue* du MOM. Le traitement est réparti sur une dizaine de serveurs physiques, sur lesquels tourne la même application, dont chaque exemplaire, chaque « instance », boucle sur le traitement:

- Recevoir un message
- Effectuer le traitement
- Acquitter le message.

Le flux de travaux est donc réparti de manière équilibrée entre les différents serveurs. Et notons que même si l'affectation est bêtement cyclique, l'équilibrage est satisfaisant puisque chaque serveur reçoit des travaux selon sa capacité à traiter.



Et l'on peut même spécialiser les *consumers*, si besoin, en leur faisant sélectionner dans la queue, les tâches qu'ils savent faire.

Topologie et réseau de brokers

Un MOM peut être constitué d'un unique broker, ou bien de différents brokers échangeant en réseau.

Selon quels critères peut-on définir ces questions de topologie ?

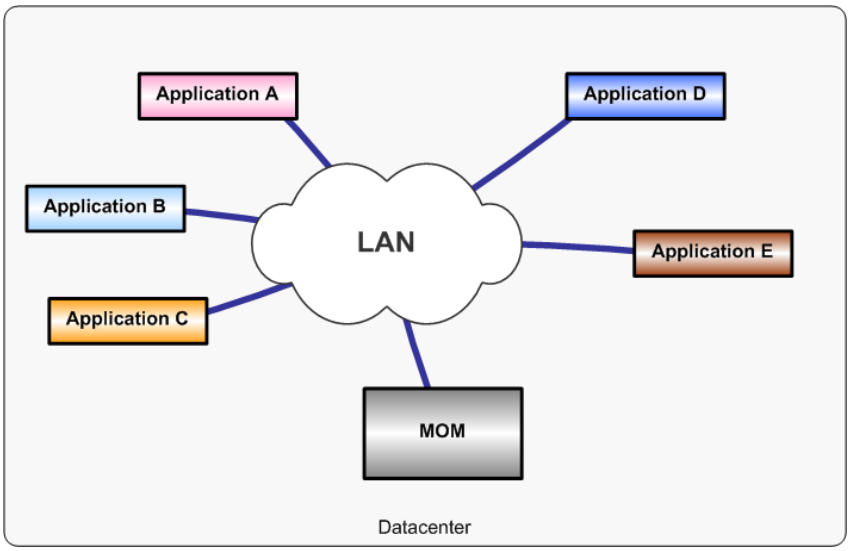
Les questions essentielles porteront sur :

- Les performances et la tenue en charge
- La tolérance aux pannes matérielles
- La tolérance aux pannes réseau

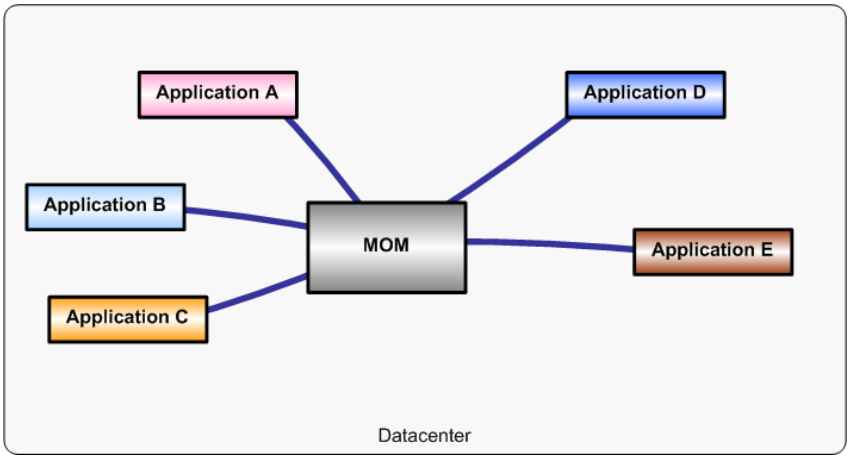
En général, sur une même plateforme, c'est-à-dire un ensemble de serveurs relevant d'un même datacenters et connectés à très haut débit, un unique broker peut suffire, pour autant que sa haute disponibilité soit assurée, et qu'il ait la capacité à traiter la volumétrie requise.

Nous verrons plus loin comment traiter la haute disponibilité. Concernant la capacité, comme nous le verrons dans les benchmarks, les MOM sont des outils construits pour de hautes performances, et un unique broker pourra acheminer plus de 1000 messages par seconde en mode persistance, et plus de 5000 sans persistance. Dans beaucoup de cas, cela peut suffire. D'autant qu'il s'agit là de débits de *traitement et d'acheminement*, il est toujours possible de confier les messages au MOM à un débit plus élevé en présence de pics.

Du point de vue réseau, on peut représenter cette configuration à un seul broker, dans un seul datacenter, simplement comme ceci :



D'un point de vue logique, on peut le visualiser comme ceci, une configuration *« hub and spoke »*, noyau et rayons :



C'est principalement lorsque les applications sont réparties sur plusieurs datacenters que l'on doit envisager des configurations à plusieurs brokers.

Rappelons que la disponibilité du MOM n'est pas juste une bonne chose, elle est *absolument fondamentale* pour les applications. Lorsqu'un utilisateur veut se connecter au site web de sa banque, on préfère bien sûr que ce site soit disponible. S'il ne l'est pas, l'utilisateur est mécontent, mais il peut ré-essayer un peu plus tard.

Pour une application s'adressant à un MOM, la question de disponibilité s'analyse différemment :

- Si le concepteur de l'application peut être certain que le MOM est toujours disponible, il ne traite pas le cas d'indisponibilité, ou plus exactement, il considère ce cas comme une erreur fatale, ou en d'autres termes : « pas de MOM, pas d'appli ». C'est souvent la politique d'une application vis-à-vis de sa base de données.

- Si au contraire l'indisponibilité du MOM est possible, le concepteur de l'application doit gérer ce cas, ce qui peut changer radicalement la logique de son application, et amener une grande complexité. L'application est-elle supposée « mettre de côté » le message en attendant le retour du MOM ? Non, certainement pas, ce serait une erreur de s'engager dans cette voie. Le MOM lui-même est déjà le moyen de « mettre de côté » le message, en cas d'indisponibilité de l'application destinataire.

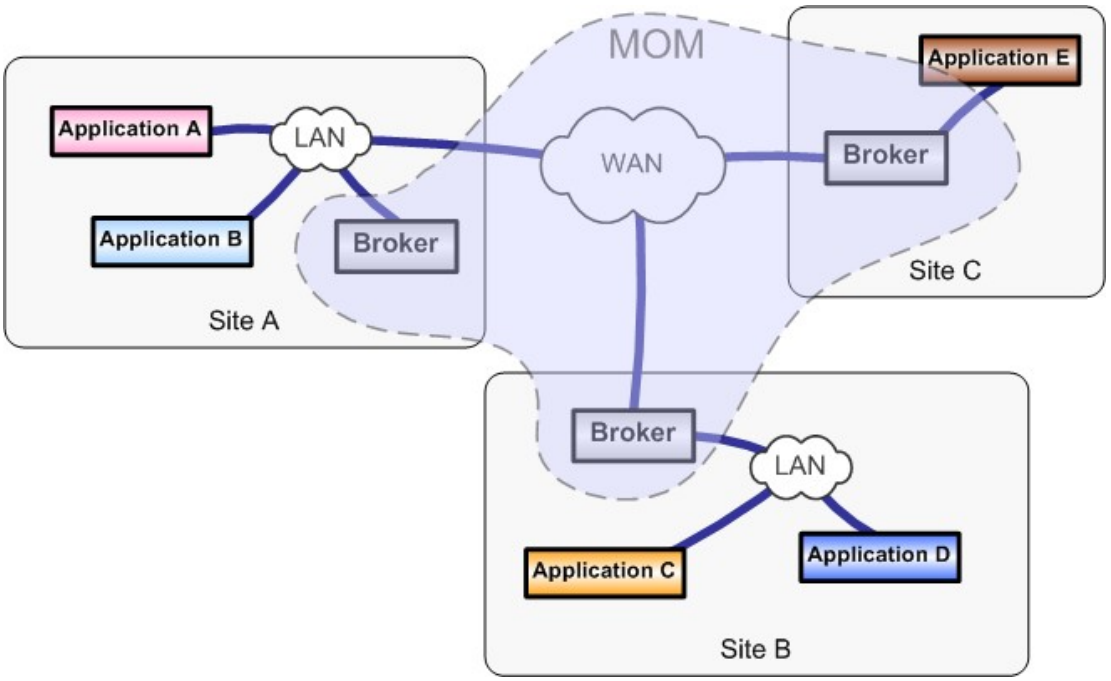
Ainsi, nous considérons qu'une application qui utilise un MOM est, le plus souvent, dans un mode où l'indisponibilité du MOM est une erreur fatale.

Et pour l'application, le MOM est indisponible lorsque le broker est indisponible ou bien n'est pas joignable.

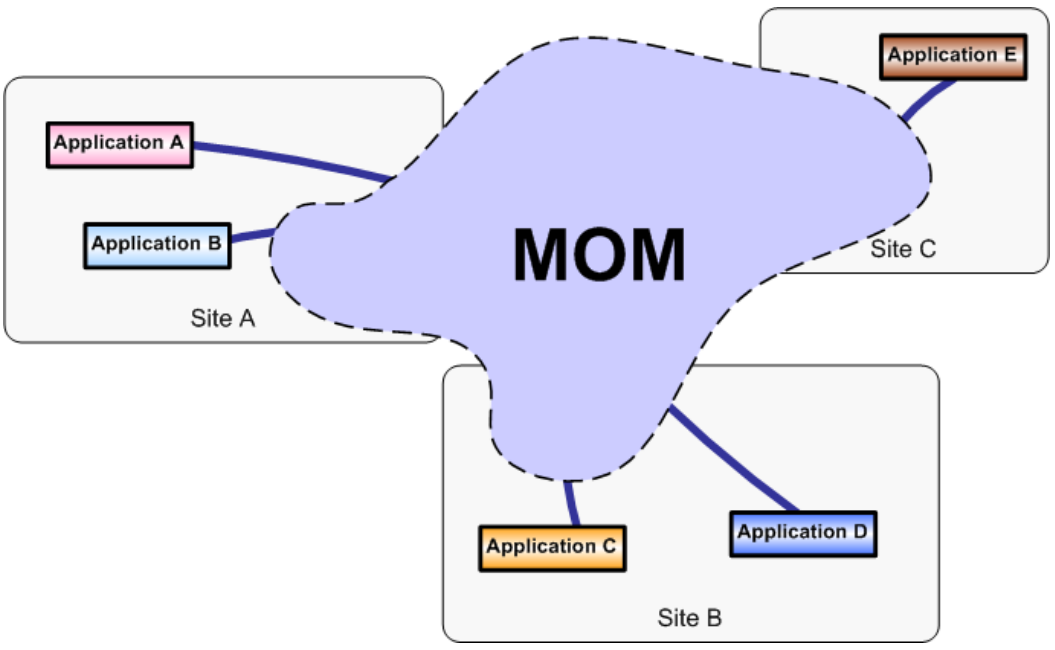
C'est pourquoi, lorsqu'un système d'information est réparti sur plusieurs datacenters, connectés en WAN, on préconise de disposer d'un broker

dans chaque datacenters. Ainsi, même lorsque la connectivité est perdue entre les datacenters, toutes les applications peuvent continuer à échanger avec le MOM, via un broker local.

Ce qui donne, d'un point de vue réseau, le modèle suivant :



Et bien sûr, d'un point de vue logique :



En présence de multiples brokers, le MOM fonctionne toujours sur un principe de « store and forward », c'est-à-dire que chaque broker conserve

les messages jusqu'à ce qu'il ait pu les transmettre à un autre broker, ceci bien sûr dans une logique transactionnelle. Les brokers échangent entre eux afin d'identifier les besoins de routage des messages. C'est-à-dire que lorsqu'une application « D » indique à son broker local qu'elle est en lecture sur telle *queue* ou tel *topic*, le broker local échange avec les autres brokers pour les informer de cette attente, et obtenir les messages de cette *queue*.

Notons qu'il n'y a pas de notion de « broker affecté à la gestion d'une queue », ni de « queue affectée à un broker », la gestion de toutes les queues est véritablement distribuée entre les brokers.

Tolérance aux pannes

Nous avons abordé plus haut, en évoquant la topologie, la question de la tolérance aux pannes réseau, aux pertes de connectivité.

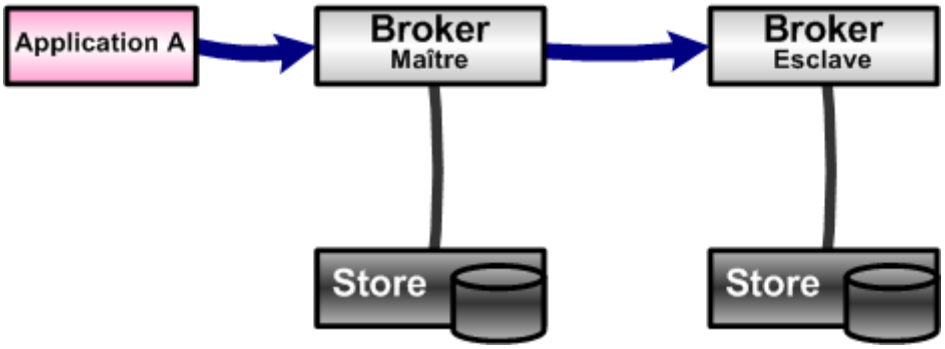
Voyons maintenant la tolérance aux pannes au niveau d'un broker particulier.

Les techniques mises en œuvre sont en fait les mêmes que pour n'importe quel serveur d'application : redondance du serveur et partage des données.

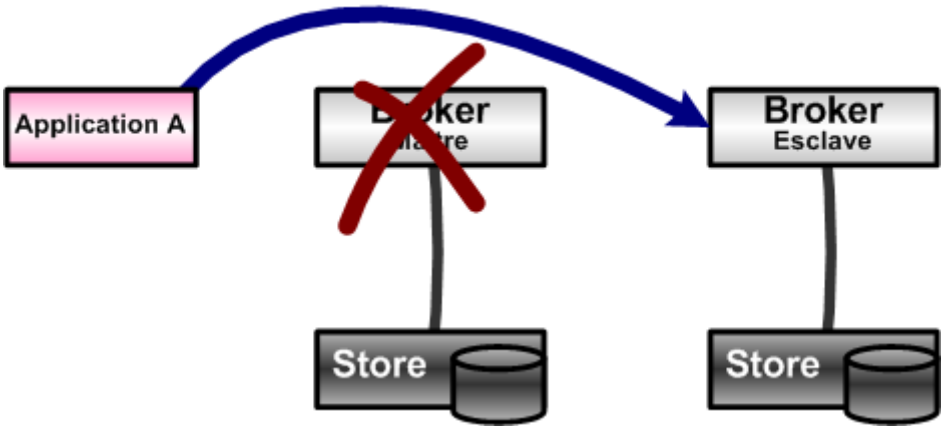
Réplication maître-esclave

Lorsqu'on met en place une réplication d'un broker maître vers un broker esclave, chaque broker possède son propre stockage, le broker maître adresse chaque message reçu à l'esclave, et le message n'est acquitté à l'application que lorsqu'il a été sécurisé sur le maître *et* sur l'esclave, c'est-à-dire que la réplication est synchrone.

On peut représenter cette configuration comme suit :



Lorsque le broker maître devient indisponible, le broker esclave reprend la fonction et toutes les applications clientes s'adressent à lui, de manière transparente.

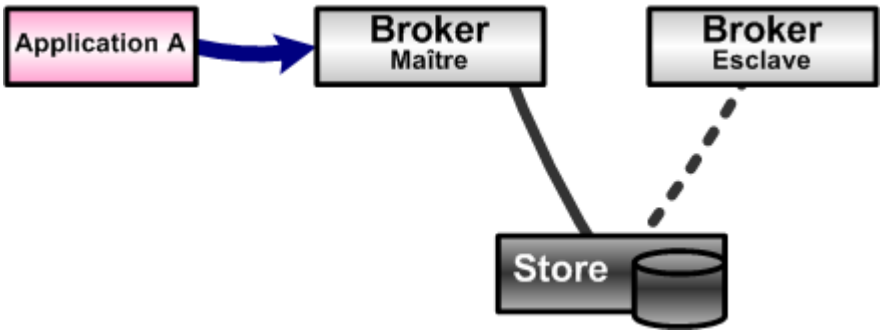


Partage du stockage

Une autre configuration possible assurer la haute disponibilité du broker est le partage du système de persistance, qu'il s'agisse d'une base de données ou bien du système de fichiers.

Dans cette configuration, il n'y a qu'un stockage, partagé entre le maître et l'esclave. Le maître détient un verrou sur une table ou un fichier, et l'esclave est en attente sur ce verrou. De sorte que lorsque le maître est arrêté, l'esclave obtient le verrou et reprend la fonction de broker principal, en accédant à tous les messages et les informations d'état qui se trouvent dans le dispositif de stockage.

On peut représenter cette configuration ainsi :



Et le dispositif peut s'étendre assez facilement à de multiples brokers esclaves.

Auto-découverte

Ces clusters de brokers sont configurables et peuvent profiter des fonctionnalités *d'auto-découverte*. Par exemple, lors de la mise en ligne d'un broker supplémentaire (configuré correctement), les brokers en cours d'exécution le reconnaitront tout de suite comme faisant partie de la plateforme.

Tous les mécanismes de découverte automatique reposent sur le *broadcast ou le multicast*. Ces dernières permettent l'envoi de paquets d'information à un ensemble de machines sur un réseau sans pour autant les avoir identifiées unitairement.

L'auto-découverte par broadcast et multicast ne fonctionne pas sur l'Internet. Dans ces cas, certains MOMs autorisent l'auto-découverte à l'aide d'un serveur d'annuaire comme LDAP. Un soin particulier doit être apporté à la sécurité de la plateforme distribuée.

LES MOMS OPEN SOURCE

Les MOMs étudiés

Nous avons sélectionné les 4 outils qui nous semblent les plus crédibles, les plus solides, et les plus pérennes, ceux sur lesquels on peut envisager sans risque de construire une architecture critique pour l'entreprise.

Les outils sélectionnés ne se différencient pas tant par la liste des fonctionnalités, qui pour l'essentiel découle de la spécification JMS. Ils se distinguent en revanche par les possibilités d'interfaçage, par des fonctionnalités avancées en particulier en matière de clustering. Ils se distinguent aussi par leur dynamique de développement, et l'estimation que l'on peut faire de leur part de marché.

Les produits sélectionnés sont les suivants :

- Active MQ
- JORAM
- Open Message Queue
- JBoss Messaging

JORAM

Présentation

JORAM ou Java Open Reliable Asynchronous Messaging, est le Middleware de consortium Object Web. Object Web est aussi connu pour son serveur d'application Java nommé Jonas **auquel est d'ailleurs intégré JORAM.**

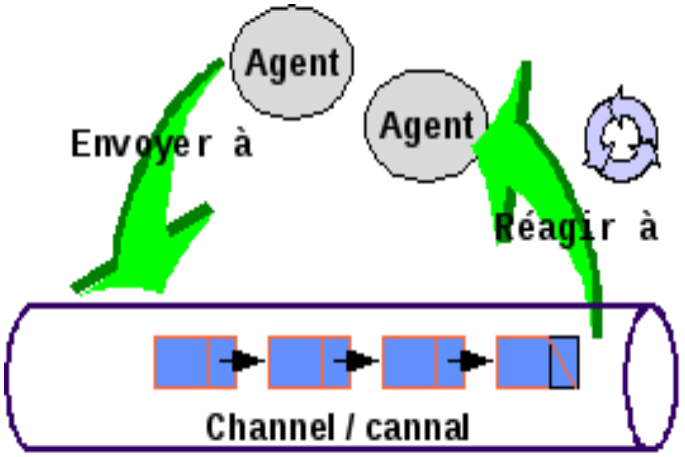
JORAM **est sortie en 1999 et est distribué sous licence LGPL depuis Mai 2000.**

Caractéristiques principales du produit

Nous allons parcourir les caractéristiques de JORAM selon les classes de fonctionnalités présentées plus haut.

Implémentation

JORAM a une architecture interne élégante, basée sur le modèle d'agent.



Architecture de JORAM

Un *agent* est un composant logiciel répondant à certains événements. Dans le cas de JORAM**, les événements sont sous forme de messages. Les** *queues* et les *topics* sont ainsi représentés par des agents. Un utilisateur connecté à la plateforme est également représenté par un agent dit *proxy*. Cette approche offre une grande flexibilité, car elle permet la création et la suppression d'agents à la volée et sur n'importe quel broker**. Un broker est donc uniquement un serveur d'agent (ou un** *container* d'agent). À l'instar des EJB, ces agents ne peuvent pas encore être déplacés de broker en broker.

Le code source récupéré du SVN JORAM **est assez bien documenté. Il est fait de «** *beans* » séparés en *Interfaces* et *Implémentations*. Dans l'ensemble, le code respecte les bonnes pratiques de développement Java**.**

Langages pris en charge

Les langages par lesquels ont peut accéder à JORAM sont :

- Java **via l'interface JMS.**
- C et C++ **: À l'aide de JNI, permettant ainsi de simuler un environnement JMS.**

Protocoles pris en charge

Le protocole interne de JORAM **est propriétaire, et n'est pas documenté. Nous estimons que c'est un handicap dans la mesure où cela tend à limiter le nombre d'environnements dans lesquels des APIs sont offertes, et à rendre plus difficiles les interconnexions. Joram le désigne simplement par « TCP », mais il est évident qu'il y a un protocole, non spécifié, au dessus de TCP/IP.**

Ainsi, JORAM ne s'appuie pas sur des protocoles standards comme AMQP ou STOMP.

JORAM **met à disposition des passerelles permettant d'étendre le nombre de protocoles gérés tout en se basant sur le protocole dit « TCP ».**

- Passerelle SOAP (grâce à un serveur d'application) : Permet la communication en SOAP avec le broker, donc en principe depuis des environnements autres que Java.

- Passerelle Mail : Cette passerelle permet d'envoyer et de recevoir des messages JMS en s'appuyant sur du SMTP (Protocole de mail). Pour cela JORAM utilise des *queues* et *topics* spécifiques. Cette passerelle est réalisée en Java.

- Passerelle FTP **: JORAM réserve des** *queues* spécifiques pour les canaux FTP. Cette passerelle fonctionne sur le même principe que la passerelle Mail**. Elle est destinée à l'échange de messages volumineux. Cette passerelle est réalisée en Java.**

Interfaces prises en charge

Selon les classes d'interface :

- Gestion des messages

JORAM **prend en charge le JMS 1.1 et est compatible avec JMS 1.0.2b. JORAM a aussi implémenté une interface JMS 1.1 destinée, au Framework J2ME, la version de l'environnement Java destinée aux mobiles, téléphones et PDAs. JORAM peut donc être mis en œuvre à partir de terminaux mobiles compatibles Java.**

JORAM **prend aussi en charge JCA 1.5, lui permettant de se connecter aux différents PGI du marché (Open ERP, …) qui le gèrent.**

- Interfaces d'Administration, Monitoring, Configuration

JORAM **supporte l'interface d'administration JMX. Il est intégrable et configurable en Java. Il supporte aussi le JAAS** pour l'authentification et les habilitations.

Gestion des messages

Outre les fonctionnalités standards, JORAM **gère :**

- La notion de hiérarchie des topics : Chaque topic peut être lié à un autre (et un seul) et recevoir tous ses messages. À son tour, le parent topic reçoit tous les messages de ces parents et les envoie à tous ses topics fils. Prenons un exemple : Imaginons trois topics : Manager, Operateurs_France, Operateurs_Espagne. On souhaite que tous les messages envoyés aux topics Opérateurs_* soient aussi envoyés au topic Manager. En plaçant Manager comme topic père aux topic Operateurs_*, tous les consommateurs recevront de façon transparente les messages envoyés aux topics Opérateurs_*.

Il n'est pas possible de faire de traitement avec JORAM**.**

Persistance des messages

La persistance peut être gérée sur le système de fichier, dans une base java embarquée (Derby, voir plus loin pour plus de détail), ou sur une base de données relationnelle externe via JDBC.

Derby est un système de gestion de base de données relationnelle embarquée. « Embarquée » veut simplement dire qu'il n'est pas nécessaire d'avoir un *serveur* de base de données, au sens d'un processus distinct. La base de données est dans le même processus que l'application. Le support de stockage de la base Derby est le fichier. Derby est une méthode avancée de lecture et d'écriture sur des fichiers.

Nous n'avons pas trouvé, dans les documentations fournies par JORAM**, d'information sur les optimisations possibles de la gestion de la persistance.**

Répartition de charge et haute disponibilité avec plusieurs sites

Comme on l'a évoqué, JORAM **est construit selon une architecture à base** *d'agent*s. Cette architecture est l'objet d'un livre blanc disponible sur le site du produit.

Grâce à son architecture, JORAM assure :

- La disponibilité : pour rappel, la défaillance d'un serveur n'affecte que les clients JMS connectés à ce serveur. Les autres continuent à fonctionner en accédant à d'autres copies du domaine. La synchronisation des *domaines* se fait d'une manière transparente, selon un principe maître-esclave.

- Répartition de charge : les applications clientes sont réparties sur plusieurs serveurs de telle sorte que la charge engendrée par la

gestion des domaines soit répartie entre les serveurs. Cette répartition peut soit être réalisée manuellement (configuration et utilisation du « Store and Forward »), soit être confiée à un load-balancer.

Interopérabilité avec d'autres MOMs

JORAM fournit un squelette de passerelle avec d'autres MOM gérant le JMS 1.1.

Gestion de la sécurité et d'un annuaire

JORAM peut être configuré pour utiliser des connexions SSL / TLS.

Il gère l'authentification et l'autorisation.

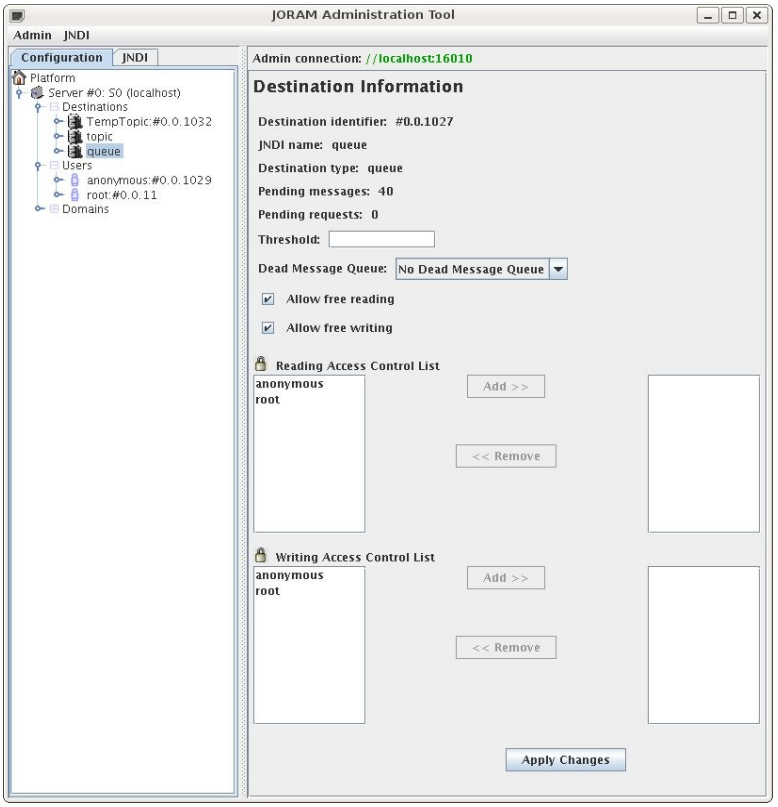
Des fichiers de configuration au format XML sont utilisés pour définir la configuration de sécurité. Il est possible également de personnaliser la gestion de la sécurité au travers JAAS.

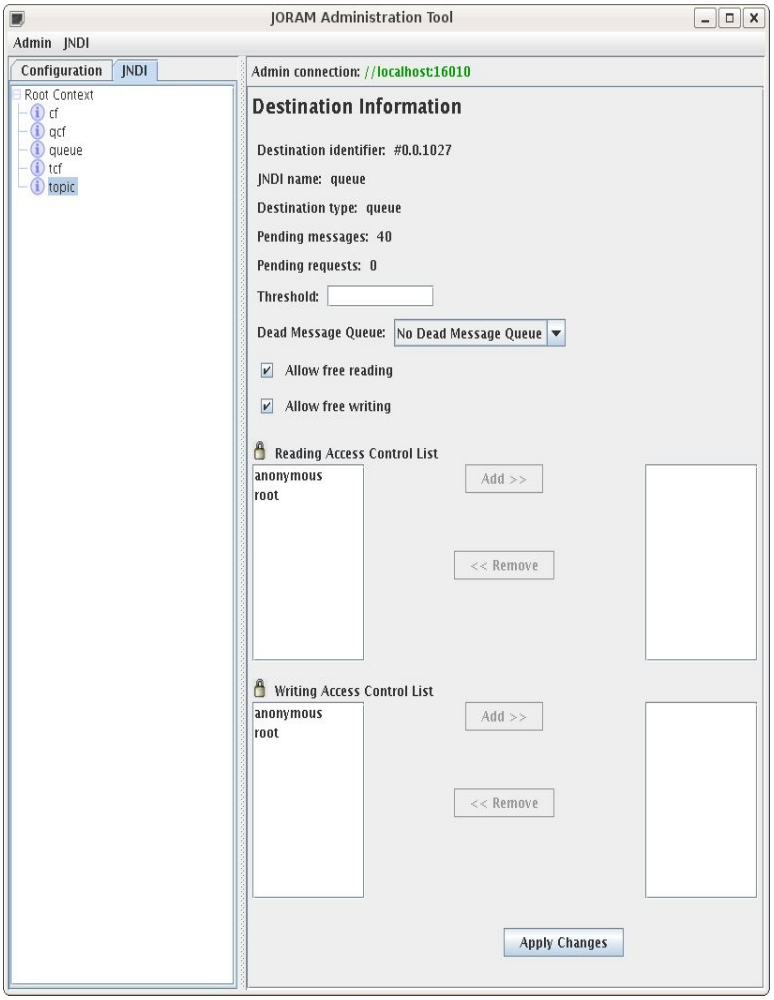
Mais ces aspects ne sont pas suffisamment documentés.

Administration

JORAM met à disposition une interface graphique d'administration. Elle se base sur l'utilisation de JMX.

Voici quelques captures d'écran de l'interface d'administration.





Lors d'une utilisation standard, l'interface d'administration graphique présente quelques problèmes. Si l'on génère beaucoup d'actions, l'application s'affole et devient erratique.

Il nous semble que cette interface devrait être surtout utilisée à des fins de démonstration.

Configuration et déploiement

Après téléchargement, et modulo l'installation d'un *runtime* Java (JRE), il suffit de quelques déclarations d'environnement pour faire fonctionner la solution.

Une vingtaine d'exemples est fournie. Un système basé sur ANT rend l'utilisation de ces exemples particulièrement simple. On regrette l'absence d'une documentation digne de ce nom concernant le C / C++ et la persistance.

La configuration du MOM se fait à l'aide de fichiers XML. Les balises XML sont assez claires. La définition d'un *broker* se fait par exemple à

l'aide d'une balise « *server* » contenant la définition de celui-ci ainsi que la définition de services.

```
<?xml version="1.0"?>
<config>
  <property name="Transaction" value="fr.dyade.aaa.util.NullTransaction"/>

  <server id="0" name="S0" hostname="localhost">
    <service class="org.objectweb.joram.mom.proxies.ConnectionManager"
             args="root root"/>
    <service class="org.objectweb.joram.mom.proxies.tcp.TcpProxyService"
             args="16010"/>
    <service class="fr.dyade.aaa.jndi2.server.JndiServer" args="16400"/>
  </server>
</config>
```

JORAM fonctionne sur tout système d'exploitation supportant au minimum Java 1.4.

Détail sur le projet

Détail

JORAM est distribué sous licence LGPL et est publié par « Object Web ». Le principal contributeur de ce projet est la startup « *ScalAgent Distributed Technologies* », une société issue à la fois de l'INRIA et de Bull.

Nous avons testé la version 5.2.1. Des mises à jour sont disponibles environ tous les 3 mois aussi bien pour les versions en cours que pour les versions antérieures.

Il n'y a pas de version commerciale de JORAM, ni de modules distribués sous une autre licence.

Qualité

JORAM utilise ANT pour gérer la construction du projet, le code source est disponible sur un SVN public. JORAM est également disponible dans le référentiel MAVEN Central qui ne contient que les binaires.

Concernant la documentation, un WIKI est hébergé sur la forge d'OW2, mais celui-ci n'est pas très riche, et surtout trop peu actualisé. La dernière mise à jour semble dater du 06/04/2006.

Un guide complet PDF en anglais abordant l'installation, l'utilisation et l'administration de JORAM est disponible sur le site. ÀA cela, s'ajoute un forum sous forme d'une mailing liste, avec accès aux archives. En moyenne, on trouve quelques dizaines de messages par mois.

Un gestionnaire de bug est présent sur la forge OW2, mais ne semble pas être utilisé par le projet, on trouve uniquement 10 anomalies entre 2003 et 2009. Le nombre de contributeurs au projet JORAM est de 24.

Le site officiel de JORAM est http://joram.ow2.org. Il a un *page rank* Google de 4, ce qui est plutôt faible pour ce genre de sites. Le site est composé d'une centaine de pages tandis que le Wiki comporte une trentaine de pages. Les archives de mails comptent, quant à elle, près 400 pages.

Le site internet de JORAM n'est pas présent sur Google Trend.

Références

Aucune référence n'est renseignée.

Active MQ

Présentation

Sorti en 2004, Active MQ est le MOM open source de la fondation Apache. Il est distribué sous licence Apache 2.0.

Active MQ s'appuie sur quelques autres projets Apache :

- Apache Camel : Implémentation partielle des « *Entreprise Integration Patterns* », que nous avons évoqués plus haut.
- Jetty : Serveur d'application Java intégré à Active MQ

Et Active MQ est à son tour utilisé par quelques autres grands projets :

- ESB : Active MQ est utilisé par plusieurs ESBs (Enterprise Service Bus) tels qu' Apache Service Mix et Mule.
- Serveur J2EE : Active MQ est intégré au serveur d'application Geronimo (certifié JEE5) comme fournisseur JMS par défaut.
- Axis et CXF : Extension de Active MQ.

Caractéristiques principales du produit

Langages d'implémentation

Le code source récupéré du SVN, ne semble pas toujours être d'une qualité exemplaire. La mise en forme du code laisse à désirer et certaines parties ne respectent pas les bonnes pratiques de codage Java : peu d'interfaces, classes et méthodes trop longues, ... Mais la robustesse du produit est néanmoins réputée.

Langages pris en charge

La diversité des langages et environnements supportés est particulièrement grande, et c'est un des grands atouts de Active MQ. Comme on l'a évoqué, l'aptitude à faire échanger des applications hétérogènes fait partie des missions naturelles d'un middleware.

Les langages à partir desquelles on peut accéder à Active MQ sont :

- C : grâce à la bibliothèque OpenWire C Client
- C++ : grâce à CMS : C'est une bibliothèque C / C++ proposant des interfaces similaires à JMS
- Ajax, RESTful et SOAP : sous condition d'utilisation des passerelles proposées par Active MQ. (La passerelle est sous forme d'un servlet Java, fonctionnant sur Jetty, ou autre)
- .Net : grâce à NMS : C'est une bibliothèque .Net proposant des interfaces similaires à JMS
- Delphi and FreePascal grâce à Habari Active MQ Client
- Perl, PHP, Pike, Python, Ruby, grâce au protocole STOMP et aux librairies client correspondantes.

On voit que le choix du duo STOMP et OpenWire comme protocole de communication a ouvert la voie à l'implémentation d'APIs dans de nombreux environnements.

De plus, s'agissant de protocoles ouverts et bien spécifiés, il est possible de réaliser un client STOMP vers ActiveMQ depuis de nouveaux environnements s'il en manquait à la liste.

Protocoles pris en charge

Les protocoles prit en charge par Active MQ sont les suivants :

- AMQP : Ce protocole est pris en charge,e mais comme sa définition est volatile, Active MQ prend en charge uniquement les versions 0.8 / 0.9
- OpenWire : Protocole de communication messages
- STOMP : Protocole de communication messages
- JXTA : C'est un protocole permettant de créer des réseaux au dessus des réseaux. JXTA (pour juxtapose), défini par une série de protocoles légers conçus pour gérer n'importe quelle application peer-to-peer. JXTA est compatible avec l'ensemble des plateformes informatiques. L'implémentation Java est basée sur du XML. Avec Active MQ, il agit en tant que connecteur.

```
jxta://hostname:port
```

- XMPP : Le protocole de messagerie instantanée utilisé par Jabber. Ainsi, on peut se connecter au MOM grâce à un client de messagerie de type Jabber.
- En ce qui concerne les protocoles proposés par des passerelles :

- Grâce aux sous-projets Axis et CXF de Apache, Active MQ gère SOAP, REST, …

Interfaces prises en charge

Selon les classes d'interface :

- Messagerie
- JCA 1.5 sous Java
- JMS **1.1 et 1.0.2b sous Java**
- NMS à partir des plateformes .Net
- CMS à partir des plateformes C/C++
- Administration, Monitoring, Configuration
- JMX**, XML, Spring, Java DSL et par messages**

Ces points seront revus plus loin.

Gestion des messages

Mis à part la gestion standard des messages imposée par la spécification JMS 1.1, Active MQ gère :

- Groupe de messages : Ceci est un concept intéressant dans la mesure où il assure que tous les messages d'un même groupe soient reçus par un consommateur déterminé. Les messages d'un groupe X seront consommés uniquement par le consommateur privilégié. Si celui-ci meurt, Active MQ choisit automatiquement un autre consommateur suivant la configuration.
- Notion de sélecteur de messages compatible avec XPATH (et SQL 92 issue de la spécification JMS)
- Cependant, il n'y a pas de notion de priorité des messages. Il est possible de la simuler en utilisant des groupes de messages ou bien des sélecteurs.
- Destination virtuelle : Il est possible de définir des *topics* et des *queues* redirigeant vers des composants du même domaine (*topic* vers *topic* et *queue* vers *queue*).
- « *Total Ordering* » : Active MQ a la possibilité d'assurer que l'ordre de réception des messages correspond bien à l'ordre d'envoi.

```
<destinationPolicy>
      <policyMap>
        <policyEntries>
          <policyEntry topic=">">
            <dispatchPolicy>
              <strictOrderDispatchPolicy />
            </dispatchPolicy>
          </policyEntry>
      </policyEntries></policyMap>
    </destinationPolicy>
```

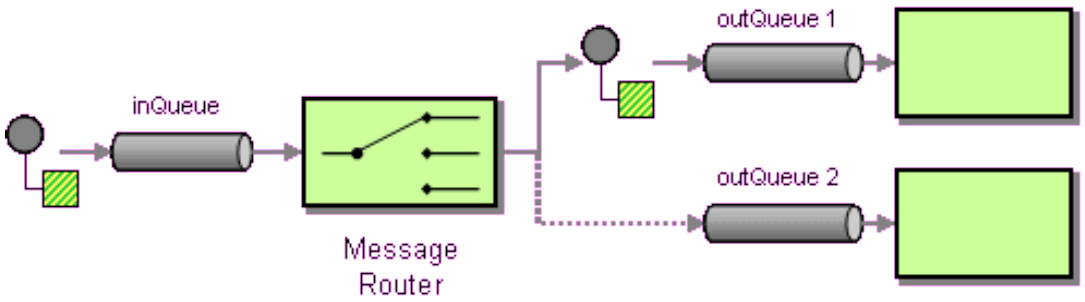
- Et bien d'autres, issues des EIP

Traitement des messages

Le traitement des messages d'Active MQ est sans doute son plus célèbre atout, après celui de sa grande connectivité. À l'aide du projet Camel qui est intégré, il a la possibilité de traiter les messages selon les modèles d'intégration d'entreprises (EIP).

Citons un exemple faisant d'Active MQ un EAI à part entière. Les fonctionnalités de routage et de transformation représentent les caractéristiques principales des EAIs.

Un exemple de routage est celui qui va rediriger le message selon son contenu.



Routage selon le contenu du message

Et la configuration Spring associée :

```
<camelContext errorHandlerRef="errorHandler" streamCache="false"
id="camel" xmlns="HTTP://camel.apache.org/schema/spring">
    <route>
        <from uri="seda:demandes"/>
        <choice>
            <when>
                <xpath>$entreprise = 'smile'</xpath>
                <to uri="seda:smile"/>
            </when>
            <when>
                <xpath>$entreprise = 'autres'</xpath>
                <to uri="seda:avant-vente"/>
            </when>
            <otherwise>
                <to uri="seda:accueil"/>
```

```
            </otherwise>
        </choice>
    </route>
</camelContext>
```

Quelques explications s'imposent. Les messages reçus sur la file « demandes » seront transmis aux files :

- smile : si la propriété entreprise du message est égale « Smile »
- avant-vente : si la propriété entreprise du message est égale « autres »
- accueil : si aucune des conditions précédentes n'est respectée.

Il faut néanmoins rappeler que Camel n'implémente pas entièrement EIP.

Gestion des transactions

Bien qu'il n'existe pas de documentation sur la méthode de gestion des transactions en interne, ActiveMQ nous donne quelques pistes.

Par exemple, la journalisation du « Message Store » permet la reprise sur incident sans perte de données lors d'un « *rollback* » (retour arrière).

Attention, par défaut, le routage et la transformation des messages ne sont pas transactionnels.

Une « Dead Message Queue » est présente. Voici un exemple de configuration :

```
<destinationPolicy>
  <policyMap>
    <policyEntries>
<!-- Set the following policy on all queues using the '>' wildcard -->
      <policyEntry queue=">">
        <deadLetterStrategy>
<!-- Use the prefix 'DLQ.' for the destination name, and make the DLQ a
queue rather than a topic -->
          <individualDeadLetterStrategy queuePrefix="DLQ."
useQueueForQueueMessages="true" />
        </deadLetterStrategy>
      </policyEntry>
    </policyEntries>
  </policyMap>
</destinationPolicy>
```

Dans cet exemple, chaque domaine aura une DMQ attribuée de manière individuelle.

Persistance des messages

Active MQ a introduit un mode de persistance appelé « Active MQ Message Store » qui joint un stockage de données sous forme de fichiers avec un système de journalisation et de mise en cache. Il affiche des performances supérieures au système de persistance sur fichier ou base

de données seule. Il affiche aussi une meilleure fiabilité, car il a été bâti pour le transactionnel.

Regardons de plus près son fonctionnement.

Lors de l'écriture d'une donnée, le message réside en cache (Memoire volatile). On construit sa référence (identification) qui sera stockée dans le journal des références. Périodiquement, une copie du journal des références caché est réalisée sur le support persistant. Ceci représente le journal des références persistant. De plus, si la donnée n'a pas été consultée depuis longtemps (configurable), elle est déplacée vers média persistant (d'une façon transactionnelle) et ses références sont mises à jour (cache et persistant).

Lors d'une lecture, on accède soit directement à la donnée en cache, soit dans le média de stockage.

Lors d'une transaction, Active MQ ne modifie que les références des messages.

Active MQ recommande d'avoir un nombre de messages inférieur à 1 million par page de cache. Le nombre de page de cache n'est pas limité.

Voici une configuration simple d'un broker utilisant l' « Active MQ Message Store » :

```
<broker brokerName="broker" persistent="true" useShutdownHook="false">
    <persistenceAdapter>
      <AMQ.PersistenceAdapter directory="Active MQ-data"
maxFileLength="32mb"/>
    </persistenceAdapter>
    <transportConnectors>
      <transportConnector uri="tcp://localhost:61616"/>
    </transportConnectors>
</broker>
```

Si un objet n'est plus référencé, il est tout simplement supprimé. Être référencé c'est être présent dans un des domaines, et donc ne pas encore avoir été consommé.

Les performances supérieures s'expliquent par le fait qu'Active MQ détecte et mesure la durée d'attente d'un message avant sa consommation. Il optimise le stockage sur un support non volatil. De fait, il ne stocke que les messages dont la durée de latence est grande.

Les supports de stockage sont compatibles avec les pilotes JDBC.

Répartition de charge et haute disponibilité multi-site.

Active MQ propose différents modes de déploiement pour une haute disponibilité :

- Cluster de brokers: permet la gestion des pannes et la répartition de la charge.

- Réseau de brokers : permet de gérer un réseau distribué de *queues* et de *topics*. Les messages seront transférés de brokers en brokers par la fonction « *store and forward* » jusqu'à ce qu'ils soient consommés. En d'autres termes, un broker recevant des messages ne correspondant à aucun domaine qu'ilhéberge, enregistrera le message et le transmettra au bon broker. L'enregistrement permet la garantie de transmission en cas d'instabilité réseau par exemple.

- Réplication en maître-esclave : permet d'avoir une redondance, cependant Active MQ supporte uniquement un esclave par maitre.

- Partage du Message Store : C'est une alternative à la réplication maître-esclave. Dans ce cas, seul le Message Store est partagé en utilisant un système de fichier sécurisé (SAN ou partage réseau) ou une base de données. La charge de traitement est répartie.

- Domaine partagé : Une application de Camel serait de partager le traitement de domaine sur plusieurs brokers. Pour ce faire, il suffit de mettre en place un domaine virtuel distribuant les messages sur plusieurs domaines.

Active MQ peut être configuré pour connaître l'emplacement des différents brokers, ou bien peut les découvrir dynamiquement tout au long du cycle de vie de la plateforme. La découverte de nouveaux brokers se fait soit grâce au broadcast, soit grâce à ZeroConf. ZeroConf est un protocole utilisant conjointement l'UDP et le Multicast.

Dès lors, la sécurité devient le point faible. Le risque qu'une personne malveillante introduise un broker malveillant pour voler ou introduire des messages est plus grand.

La découverte de machine peut aussi se faire par l'intermédiaire d'un annuaire du type LDAP. Un broker mis en ligne se déclare dans un annuaire. Les autres machines connectées à l'annuaire se rendent compte de l'apparition d'une nouvelle machine et communiquent avec lui.

Un exemple de configuration de la découverte par LDAP:

```
[...]
<networkConnectors>
      <ldapNetworkConnector uri="ldap://myldap.mydomain.com:389"
                            base="dc=brokers-for-srv-a,dc=mydomain,dc=com"
                            anonymousAuthentication="true"
                            searchFilter="(cn=*)"
                            searchScope="SUBTREE_SCOPE"
                            networkTTL="2"
                            />
```

```
</networkConnectors>
[...]
```

Interopérabilité avec d'autres MOMs

Active MQ fournit une passerelle JMS aisément configurable (DSL, Spring XML). L'authentification est aussi prise en compte par les fichiers de configuration. Ces fichiers de configuration peuvent être intégrés à ceux d'Active MQ.

Gestion de la sécurité et d'un annuaire

L'authentification et la gestion des droits sont intégrées sous forme de plugins dans Active MQ. Les plugins proposés par défaut s'appuient sur JAAS ou sur des fichiers XML.

L'exemple le plus simple est le suivant :

```
[...]
<simpleAuthenticationPlugin>
        <users>
                <authenticationUser username="system" password="manager"
                        groups="users,admins"/>
                <authenticationUser username="user" password="password"
                        groups="users"/>
                <authenticationUser username="guest" password="password"
groups="guests"/>
        </users>
</simpleAuthenticationPlugin>
[...]
```

L'interconnexion entre brokers peut aussi être sécurisée par mot de passe et / ou chiffrement (SSL).

```
[...]
<networkConnectors>
   <networkConnector name="brokerAbridge"
                     userName="user"
                     password="password"
                     uri="static://(SSL://brokerA:61616)"/>
</networkConnectors>
[...]
```

Il est possible d'encapsuler les connexions dans du SSL entre les clients et un broker pour sécuriser les échanges. Le SSL se comporte donc comme un connecteur à part entière.

Il possible de lier la sécurité de la plateforme avec un serveur LDAP.

Active MQ fournit une interface de personnalisation via des « Interceptors ». Il est un ainsi possible d'étendre les possibilités de Active MQ très facilement. L'exemple le plus commun serait la gestion de l'authentification. Les « Interceptors » permettent de modifier certains comportements internes sans changer le cœur d'Active MQ et en compatibilité avec les versions futures.

Administration

Le monitoring et l'administration de la plateforme sont proposés :

- à travers de l'interface JMX
- au moyen d'une interface web (web console)
- par des messages : cette fonctionnalité est aussi disponible à distance via le protocole XMPP (Voir le Glossaire).

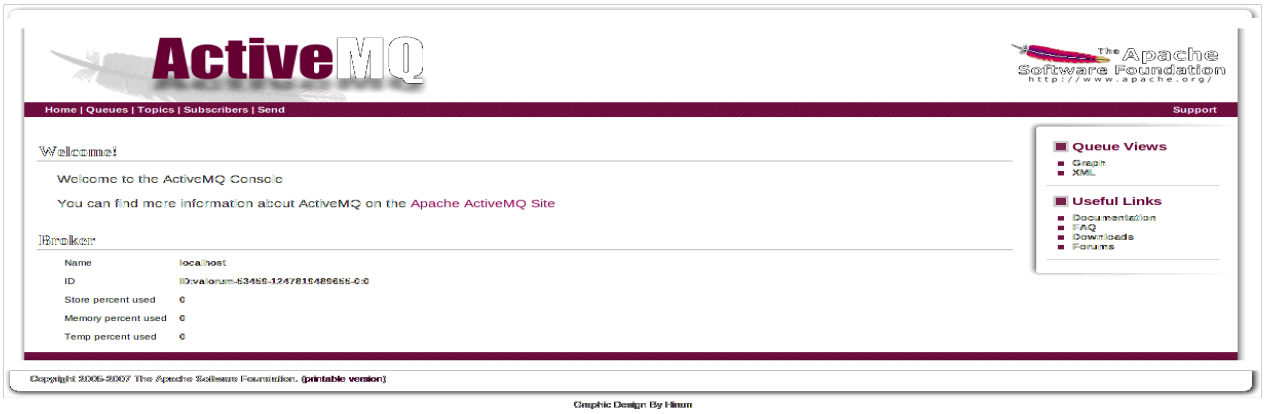
Active MQ propose des « Advisory Message » (message d'information) qui permettent de connaître l'état du système. Voici des exemples de métriques :

- les connexions clients
- les files d'attentes créées et détruites par les applications
- les messages expirés
- …

Les « Advisory Messages » sont organisés en queues et topics protégés par mot de passe. On peut y accéder à partir d'un simple client Active MQ (JMS ou autre).

Active MQ implémente aussi des « *Mirrored Queues* » : les messages envoyés à une file d'attente seront, de manière transparente, envoyés sur un Topic. Même si cette fonctionnalité est à utiliser avec précaution, elle permet à un ou plusieurs clients de suivre l'état d'une file d'attente. C'est l'application du design « *Wire Tap* » (les écoutes téléphoniques pratiquées par les espions) de EIP.

De plus, Active MQ nous fournit une interface d'administration Web. Cette interface est démarrée par défaut à l'aide de Jetty. Elle démarre par défaut à l'adresse suivante : HTTP://0.0.0.0:8161/admin



Capture d'écran de l'administration d'Active MQ

Configuration et déploiement

Active MQ peut être installé sur n'importe quelle plateforme supportant au minimum Java 5.

Active MQ est configurable en utilisant des fichiers XML intégrables à Spring. Active MQ se configure aussi à l'aide de Java DSL.

Active MQ peut aussi être configuré et lancé à partir d'un autre programme (Java), c'est la notion de « Embedded Broker » : le broker n'est plus un processus indépendant auquel le programme s'adresse par le réseau, il tourne dans le même processus que le programme client.

Active MQ est livré avec un ensemble d'exemples codés en Java ou en Ruby. Tous les cas d'utilisation d'Active MQ ne sont pas couverts par la trentaine d'exemples fournis.

Détail sur le projet

Détail

Active MQ a été principalement développé par la société LogicBlaze, et racheté par IONA technologies en 2007. IONA technologies était célèbre dans les années 90 pour son expertise CORBA.

La dernière version d'Active MQ est la 5.2.0, mais la version 4.1.x est encore maintenue par Active MQ.

Active MQ n'a pas de version commerciale.

Qualité

Active MQ utilise MAVEN pour gérer le projet. Le code source est disponible sur un SVN public dans lequel on retrouve la branche de développement, mais aussi chaque version depuis la 4.0. À noter que le projet est également disponible dans le référentiel MAVEN central.

Le site web du projet propose une documentation détaillée et utile. Certains des exemples que nous citons sont issus du site. On remarque cependant la présence de fautes d'orthographe ainsi que de nombreuses pages « en cour de construction ». Un forum pour les utilisateurs d'Active MQ est disponible sur lequel on recense une centaine de sujets par mois. À cela, s'ajoute un « bug tracker » (JIRA) contenant les différents bugs référencés par version.

Le projet possède 114 contributeurs dont une trentaine y travaille à temps plein.

Le site officiel d'Active MQ est http://activemq.apache.org. Il possède un page Rank de 8, ce qui reflète la forte popularité de l'outil. Google référence à peu près 14 200 pages.

La communauté dispose d'un site officiel, sous forme de Wiki. Elle a aussi une mailing liste, un forum et un salon IRC. Le temps de réponse moyen est de l'ordre de 2 jours. Remarquons que certaines questions ne trouvent pas de réponses.

Signalons aussi le livre « Active MQ in Action », aux éditions MEAP. Ce livre, disponible uniquement en version anglaise, est une bonne lecture pour appréhender et utiliser Active MQ. Son existence même témoigne de l'intérêt suscité par le produit.

Références

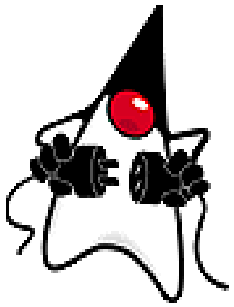
Active MQ est utilisé par de nombreux projets faisant partie de la fondation Apache (Geronimo, Service Mix, Jet Speed, Apache Directory), mais également par des projets extérieurs à cette fondation (Eclipse, Active Cluster, Mule, Open IM).

Aucune autre référence client n'est indiquée sur le site.

MOM Open Message Queue (OMQ)

Présentation

OMQ est le Middleware Orienté Message de Sun. Il a été développé pour fonctionner conjointement avec GlassFish (Open ESB).



Le principal contributeur est la communauté Sun / Java.

OMQ a été réalisé pour fonctionner avec GlassFish, le serveur d'application de Sun. Cependant, OMQ peut facilement fonctionner tout seul ou avec d'autres types de serveur d'application Java.

OMQ est distribué sous deux licences : CDDL ou GPL v2.

Caractéristiques principales du produit

Langages d'implémentation

Les sources, récupérables du site Internet de la solution sont mal organisées. D'une part, on constate la présence de binaires, de fichiers C et autres. De plus, il n'y a pas de système de compilation automatique du type MAVEN ou ANT.

Sun fournit néanmoins une documentation indiquant comment compiler la solution (via NetBeans).

On remarque la présence de répertoires (à la racine de src) nommés Solaris et Win32 (Windows), référent à des bouts de code spécifique. Qui a dit que Java était multiplateformes ?

Dans cet amas de fichiers, on retrouve même le code de l'interface d'installation.

En ce qui concerne le code java en lui même, il est bien documenté et semble respecter les standards.

Langages pris en charge

Les seuls langages pris en compte sont :

- Java via JMS 1.1 ill ne gère pas le JMS 1.0.2)
- C : l'API est propriétaire Java, sa spécification est documentée par Sun à l'adresse suivante :
  http://docs.sun.com/app/docs/doc/819-7756

On note le petit nombre de langages pris en compte, ce qui constitue une faiblesse.

Protocoles pris en charge

Les protocoles externes pris en charge sont les suivants :

- *UMS comme Universal Messaging System* : C'est un protocole de communication comparable à AMQ**P.** Sun ne le met guère en avant, étant données ses limitations en termes fonctionnalités et de performance. UMS est basé sur du XML, ce qui alourdit un peu les échanges. Sun fournit sa spécification à l'adresse suivante : HTTPS://mq.dev.java.net/4.3-content/ums/umsIntro.html

A l'aide de passerelles, OMQ gère aussi le :

- SOAP : sur un support HTTP à partir d'un serveur d'application.
- HTTP : passerelle sur un serveur d'application.

Il y a donc peu de protocoles pris en compte.

Le protocole interne d'OMQ n'est pas documenté.

Interfaces prises en charge

Selon les classes d'interface :

- Messagerie
  - JCA 1.5 sous Java
  - JMS **1.1 sous Java**
  - API C : Elle est propriétaire à Java**, sa spécification est documentée par Sun à l'adresse suivante :** HTTP://docs.sun.com/app/docs/doc/819-7756
- Administration, Monitoring et configuration
  - JES : Java Monitoring plateforme Support
  - JAAS

Gestion des messages

OMQ ne gère pas la priorité des messages.

OMQ gère la compression et la décompression des messages à la volée. Un exemple :

```
MyMessage.setBooleanProperty("JMS_SUN_COMPRESS",true);
```

Une des nouvelles fonctionnalités originales est la gestion des « WildCard Topics ». En autorisant l'utilisation d'une syntaxe particulière, OMQ autorise l'envoi d'un message à plusieurs domaines. Un exemple simple est l'envoi d'un message vers toust les Topics. Pour ce faire, on envoie un message vers un topic se nommant « * ». Voici quelques autres exemples :

Tableau 1 : Exemple de WildCard

Paterns	Résultats
*.sun.com	Retourne toute chaine de caractère finissant avec .sun.com
(quark|energy).sun.com	Retourne soit quark.sun.com ou energy.sun.com
.	Retourne toute chaine de caractère ayant un point au milieu.

Pour finir avec la gestion des messages, OMQ gère la validation des contenus XML : « *XML Schema Validation* ».

Traitement des messages

Le traitement des messages n'est pas pris en compte par OMQ.

Gestion des transactions

La gestion de transaction est offerte à la fois à partir du C et du Java. OMQ propose aussi des interfaces du type XA / JTA.

La gestion interne des transactions n'est pas spécifiée.

Persistance des messages

Il est possible de réaliser de la persistance sur le système de fichier. La persistance est aussi disponible dans des bases de données telles que : Oracle, MySQL, PostgresSQL, Java DB (Derby), toutes accédées via JDBC.

Répartition de charge et haute disponibilité multi site.

Deux modes sont disponibles :

- Cluster Normal : Ce mode n'oblige pas la persistance. Il permet la répartition manuelle de la charge sur plusieurs brokers. La disponibilité de la plateforme se trouve aussi améliorée. Cependant, si un ou plusieurs brokers venaient à mourir, leurs messages seraient perdus.

- High-availability (Haute disponibilité) : En introduisant de la persistance, OMQ améliore encore la fiabilité de la plateforme. Même si tous les brokers meurent au même moment, aucun message ne sera perdu (les messages ayant été pris en compte). Cette solution amène de moindres performances.

OMQ ne gère pas de réplication Master/Slave.

Il n'y a pas de système de découverte automatique de broker.

OMQ se montre ainsi limité concernant les problématiques d'entreprise.

Interopérabilité avec d'autres MOMs

Sun ne fournit aucun bridge JMS ou autre. Il est ainsi à notre charge d'en créer ou d'en adapter un (open source) à nos besoins.

Gestion de la sécurité et d'un annuaire

OMQ support SSL / TLS comme mode d'encryptions des messages.Celui-cii peut se placer aussi bien entres applications et brokers qu'inter-brokers.

Les applications clientes (consommatrices ou productrices) peuvent se connecter grâce à un couple (nom d'utilisateur, mot de passe). Les mots de passe sont encodés à l'aide de l'algorithme MD5.

OMQ gère les groupes d'utilisateurs. On peut personnaliser les accès aux éléments des brokers (queues, topics, administration, monitoring) par utilisateurs ou par groupes.

Les supports de stockage des éléments de sécurité sont:

- Fichier de configurations sous format XML

- LDAP

Les propriétés suivantes contrôlent le comportement d'OMQ vis-à-vis du LDAP

```
imq.user_repository.ldap.server
imq.user_repository.ldap.principal
imq.user_repository.ldap.password
imq.user_repository.ldap.propertyName
imq.user_repository.ldap.base
imq.user_repository.ldap.uidattr
```

```
imq.user_repository.ldap.usrfilter
imq.user_repository.ldap.grpsearch
imq.user_repository.ldap.grpbase
imq.user_repository.ldap.gidattr
imq.user_repository.ldap.memattr
imq.user_repository.ldap.grpfilter
imq.user_repository.ldap.timeout
imq.user_repository.ldap.ssl.enabled
```

La gestion de l'authentification et de l'autorisation peut être personnalisée à l'aide de l'API JAAS.

Administration

OMQ fournit aussi des outils d'administration en ligne de commande permettant, à l'aide de scripts (shell ou autres), d'automatiser certaines tâches. A titre d'exemple, « imqadmin » et « imqcmd » permettent de gérer un parc de brokers, de recharger une nouvelle configuration, … Ces outils se montrent ainsi particulièrement utiles.

Un monitoring du middleware est possible par messages. Il suit les mêmes concepts que les « Advisory Messages » d'Active MQ.

La plateforme OMQ implémente JMX.

Configuration et déploiement

OMQ est réalisé en Java. Voici la liste des systèmes d'exploitation dont Sun annonce le support :

- Solaris 9 ou 10
- RedHat Entreprise Linux Advanced/ Entreprise Server
- Windows XP / 2000 Server / 2009 Server

Le fonctionnement sur une Linux Debian semble tout à fait satisfaisant.

Toujours selon Sun, OMQ peut aussi bien tourner sous une architecture Sparc que x86. Il requiert un minimum de 256 Mo de RAM, mais Sun recommande 2 Go de Ram pour de la HA ou pour de gros volumes de messages.

Lors du téléchargement du paquet du site de Sun, on remarque la présence d'un installateur graphique.

En ce qui concerne les exemples, ils sont au nombre de 41, illustrant : JMS, JMX, le monitoring, et SOAP. On constate aussi la présence d'une dizaine d'exemples montrant l'utilisation de l'API C. Les exemples se limitent à l'utilisation des services de messageries et de monitoring d'OMQ. Dommage qu'aucun exemple ne montre la mise en place d'une plateforme en cluster ou high-availability (haute disponibilité).

Un autre point regrettable est que le lancement des exemples est à faire manuellement tout en manipulant le classpath du compilateur et de la VM.

La configuration de la solution se fait grâce à des fichiers non –XML dont voici un exemple :

```
imq.cluster.brokerlist=host1:9876,host2:5000,ctrlhost
```

Cette ligne informe OMQ de la liste des brokers disponibles.

Il est vraiment plus simple et pratique d'utiliser les scripts fournis que de remplir les fichiers de configuration, ce qui est bien dommage.

Détail sur le projet

Détails

La version d'OMQ étudié est la 4.3. On remarque une assez conséquente liste de bugs dont certains sont particulièrement gênants :

- Impossibilité de parcourir une queue qui est gérée par un autre broker (Browse).
- La persistance avec HADB est limitée en nombre de messages (10 000) et en taille (10 Mo).

… la liste complète des bugs est à cette adresse : HTTP://docs.sun.com/app/docs/doc/820-6360/aembi?a=view

Il est possible d'obtenir une version commerciale. Cependant, aucun détail n'est fourni.

Qualité du projet

Sun fournit un Wiki contenant des exemples de code. La communauté dispose d'un Forum, une mailing-list ainsi qu'un système de gestion des tickets.

Le site prend parfois plus de *3 secondes* à s'afficher. Il nous est arrivé à plusieurs occasions que le site ne soit plus disponible. Tous ces défauts suggèrent que le projet n'est pas très actif, ou en déperdition.

Toutefois, si l'on s'intéresse à la réactivité des contributeurs, on constante un délai moyen de réponse de l'ordre de l'heure, ce qui est fort appréciable.

Sur le bug tracker, il existe encore des bugs ouverts depuis près d'un an, et de même certaines questions sur le forum n'ont pas trouvé de réponse depuis plusieurs mois.

Selon Google, le site officiel HTTP://mq.dev.java.net est constitué de 62 pages et obtient un *page rank* de 6.

Références

Aucune information n'est donnée sur les clients ou utilisateurs d'OMQ.

MOM JBoss Messaging (JBM)

Présentation

JBoss a donné naissance à *JBoss Messaging (JBM)* devenu ensuite *JBoss Queue (JBQ)*, actuellement en sa version 1.4.0 SP3.

Dès 2006, JBM a été réalisé dans l'idée d'une intégration avec les produits JBoss. Il peut, modulo d'assez lourdes manipulations, fonctionner en mode « standalone ».

La filiation à RedHat lui confère une place particulière parmi les middlewares d'entreprise Open Source, et d'autant plus qu'il est sous licence LGPL.

JBM a été réalisé, comme son nom l'indique, par la communauté JBoss et RedHat, leader mondial dans le domaine de l'open source.

Caractéristiques principales du produit

Langages d'implémentation

Récupéré à partir du site Internet de JBM, le code source est assez bien organisé. Un système de compilation automatique du type MAVEN est présent.

Quant aux sources, elles ne sont pas toujours bien formatées. La documentation du code est à revoir sérieusement et certaines méthodes sont vraiment trop volumineuses.

Langages pris en charge

Le seul langage de programmation pris en charge par JBoss est le Java, et ceci par l'intermédiaire de l'API JMS.

Des quatre MOMs de notre sélection, il est celui qui présente le moins de connectivité.

Protocoles pris en charge

JBM ne gère qu'un seul protocole externe dont la documentation est introuvable. Toutefois, la *roadmap* du projet indique que l'outil compte implémenter STOMP.

La version 2 de JBM, en version Beta implémente déjà le protocole AMQP.

Des quatre MOMs comparés ici, il est aussi le plus pauvre dans cette catégorie.

Interfaces prises en charge

Selon les classes d'interface :

- Messagerie
  - JMS 1.1 : depuis du Java
  - JCA 1.5 : depuis du Java
- Administration, Monitoring et configuration
  - JAAS : depuis du Java
  - JMX : depuis du Java

Rien de bien nouveau.

Gestion des messages

JBM gère la priorité des messages. JBM réorganise l'ordre de délivrance des messages suivant leur priorité.

JBM ne gère ni la hiérarchie des messages ni le concept de groupe de messages.

Traitement des messages

Les modifications à la volée des messages ne sont pas prises en compte par la solution de JBoss.

Il est possible, de programmer l'envoi de message, c'est-à-dire de définir une propriété particulière qui ordonne au broker de rendre un message disponible à une heure donnée.

Gestion des transactions

Le comportement de la DMQ est standard.

La gestion interne des transactions n'est pas précisée.

Persistance des messages

JBM support plusieurs médias de stockage : Hypersonic, Oracle, Sybase, MS SQL Server, Postgres et MySQL. Ils sont tous compatibles JDBC.

Par défaut, c'est Hypersonic qui est choisi. Une note de JBoss fait remarquer que Hypersonic ne devrait pas être utilisé en production à cause :

- de sa gestion limitée des transactions.
- de son mauvais comportement à forte charge.

Un exemple de configuration pour Hypersonic est :

```
 <mbean code="org.jboss.messaging.core.JMX.JDBCPersistenceManagerService"
      name="jboss.messaging:service=PersistenceManager"
      xmbean-dd="xmdesc/JDBCPersistenceManager-xmbean.XML">
      <depends>jboss.JC
A:service=DataSourceBinding,name=DefaultDS</depends>
      <depends optional-attribute-
name="TransactionManager">jboss:service=TransactionManager</depends>
      <attribute name="DataSource">java:/DefaultDS</attribute>
      <attribute name="CreateTablesOnStartup">true</attribute>
      <attribute name="UsingBatchUpdates">true</attribute>
      <attribute name="SqlProperties"><![CDATA[
      CREATE_DUAL=CREATE TABLE JBM_DUAL (DUMMY INTEGER, PRIMARY KEY
(DUMMY)) ENGINE = INNODB
      CREATE_MESSAGE_REFERENCE=CREATE TABLE JBM_MSG_REF (CHANNEL_ID
BIGINT, MESSAGE_ID BIGINT, TRANSACTION_ID BIGINT, STATE CHAR(1), ORD
BIGINT, PAGE_ORD BIGINT, DELIVERY_COUNT INTEGER, SCHED_DELIVERY BIGINT,
PRIMARY KEY(CHANNEL_ID, MESSAGE_ID)) ENGINE = INNODB
      CREATE_IDX_MESSAGE_REF_TX=CREATE INDEX JBM_MSG_REF_TX ON JBM_MSG_REF
(TRANSACTION_ID)
      CREATE_IDX_MESSAGE_REF_ORD=CREATE INDEX JBM_MSG_REF_ORD ON
JBM_MSG_REF (ORD)
. . .
      SELECT_EXISTS_REF_MESSAGE_ID=SELECT MESSAGE_ID FROM JBM_MSG_REF
WHERE MESSAGE_ID = ?
      UPDATE_DELIVERY_COUNT=UPDATE JBM_MSG_REF SET DELIVERY_COUNT = ?
WHERE CHANNEL_ID = ? AND MESSAGE_ID = ?
      UPDATE_CHANNEL_ID=UPDATE JBM_MSG_REF SET CHANNEL_ID = ? WHERE
CHANNEL_ID = ?
      LOAD_MESSAGES=SELECT MESSAGE_ID, RELIABLE, EXPIRATION, TIMESTAMP,
PRIORITY, HEADERS, PAYLOAD, TYPE FROM JBM_MSG
. . .
      ]]></attribute>
      <attribute name="MaxParams">500</attribute>
      <attribute name="UseNDBFailoverStrategy">true</attribute>
   </mbean>
```

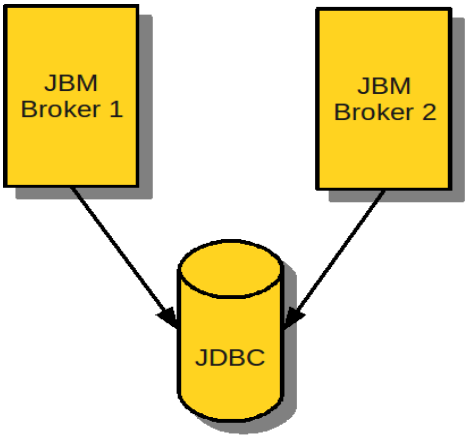
La partie du milieu remplacée par des « . . . » représente la définition des commandes SQL correspondant aux actions possibles du broker. À des fins d'optimisation par exemple, il est possible de personnaliser très finement la gestion de la persistance par le MOM.

Répartition de charge et haute disponibilité multi-site

JBoss garantit l'entière compatibilité avec une architecture en cluster aussi bien pour le mode point à point que le mode par abonnement.

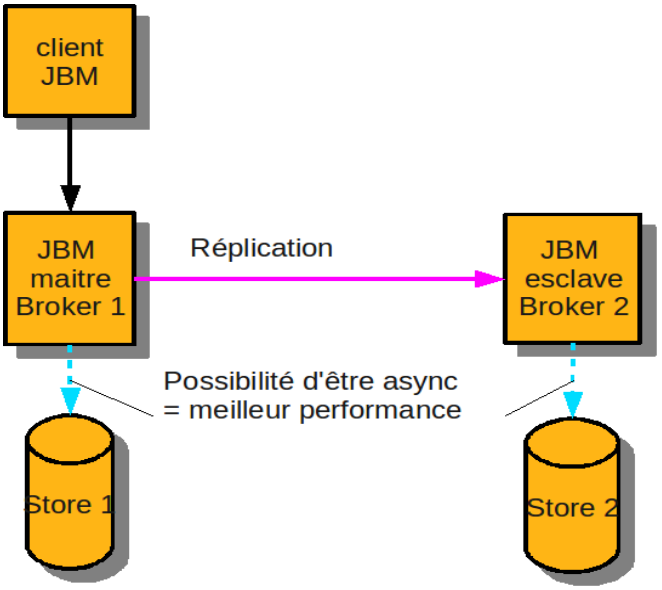
Les messages peuvent être amenés à être routés de cluster en cluster considérant la charge de chaque machine ainsi que leur performance : c'est le concept de « Store and Forward ».

Il est aussi possible de partager une même base de données.



Partage de la base de données

JBM gère aussi la découverte automatique de brokers par multicast. La réplication en maitre esclave est aussi gérée.



Faisons une petite note sur la synchronisation des différents « stores » (média de stockage). Une gestion synchrone implique la mise en attente du client JMS jusqu'à confirmation de l'écriture sur les deux brokers. Suivant le nombre et la taille des messages, ce processus peut prendre plus ou moins de temps. À l'inverse une gestion asynchrone signifie que le broker maître réagira comme s'il n'y avait pas de broker esclave et rendra la main au client dès la prise en compte du message.

Interopérabilité avec d'autres MOMs

La configuration de JBM est fournie avec une passerelle JMS. Le site de JBoss nous propose d'ailleurs un tutoriel portant sur la configuration d'une passerelle entre JBoss MQ et JBM (JBoss Messaging), car l'un et l'autre n'utilisent pas le même protocole interne.

Le protocole interne n'est pas ouvert, limitant l'accès direct à la plateforme MOM.

Gestion de la sécurité et d'un annuaire

Selon les spécifications de JBoss, la sécurité est gérée par JBM à l'aide de fichiers de configuration. Par ailleurs, elle peut être personnalisée par JAAS.

La gestion de la sécurité est réalisée par utilisateurs et par rôles. Par défaut, les informations d'authentification sont issues de fichiers XML. Le MOM a aussi la possibilité de se connecter à un LDAP.

Le chiffrement des données à l'aide de SSL / TLS est aussi supporté.

Administration

La plateforme JBM implémente JMX. Cependant, aucune interface graphique n'est fournie.

JBM a introduit quelques spécificités au niveau de l'implémentation du JMS. Sans dénigrer la spécification 1.1, JBM rajoute quelques sucreries. Par exemple, on peut maintenant récupérer les statistiques sur la plateforme sans passer par JMX. (Méthode intitulée « Message Counter »). Rappelons néanmoins que l'utilisation de ces fonctions supprime un avantage majeur de JMS, la portabilité du code. À utiliser avec modération.

Configuration et déploiement

L'utilisation de JBM à travers JBoss Application Server (JAS) est très aisée. Il suffit de les télécharger (JAS + JBM), de configurer les variables d'environnement et d'exécuter un script de configuration situé dans le dossier JBM.

Notons toutefois que l'installation de JBM en « standalone » est une opération assez lourde et n'est vraiment pas dans « l'esprit JBoss ». On comprend, après utilisation de l'outil qu'il est vraiment intégré à JAS. Par exemple, les fichiers de configuration, sous forme de XML, sont complètement intégrés à JBA.

Une trentaine d'exemples sont fournis. Ils traitent entre autres de : Passerelle JMS, Clustering, Web Service, Reprise sur erreur, le chiffrement des transmissions.

Détail sur le projet

Qualité du Projet

Malgré sa jeunesse, la communauté de développeur JBM dispose d'un SVN, un forum et un service de suivi de tickets d'incidents.

Les utilisateurs ont, quant à eux : un Wiki et un forum. Ils sont souvent indisponibles.

Le site officiel de JBM possède 17 Pages (HTTP://www.jboss.org/jbossmessaging/)

Son page Rank est de 6.

Un support technique est disponible via mail, chat (IRC) et forum. Aucun support commercial n'est disponible pour ce produit. Cependant, les produits JBoss intégrant JBM possèdent quant à eux un support commercial via email uniquement.

Une équipe de 4 personnes s'occupe à plein temps du projet.

Un Wiki **et des documentations sont fournis par la communauté JBoss.**

Références

Le site internet de JBoss présente les entreprises qui ont adoptées leurs produits, qui incluent Enernoc, Scania, Iwbank, Covad, AcXium.

Autres

La version 2.0 de JBM est en préparation dans les « bunker top secret » de la communauté JBoss. Cette version apportera des nouveautés par lesquelles:

- AMQP / STOMP
- Conception basé sur POJO
- Gestion de gros messages (exemple : 8 Go)
- …
- À partir de cette version, il s'inscrira dans la catégorie des concurrents sérieux d'Active MQ

Comparatif

		JORAM	AMQ	OMQ	JMQ
Langage	Java	★★★	★★	★★★	★★
Protocoles	Interne (non documenté / documenté / ouvert)	★	★★★	★	★
	AMQ.P	✗	★★	✗	✗
	Open Wire	✗	★★★	✗	✗
	STOMP	✗	★★★	✗	✗
Passerelles fournies	SOAP	★★	★★	★★	✗
	RestFul	✗	★★	✗	✗
	Mail	★★	★★	✗	✗
	FTP	★★	★★	✗	✗
	JavaScript / Ajax	✗	★★	✗	✗
Interfaces	JMS 1.0.2b	★★	✗	✗	★★
	JMS 1.1	★★★	★★★	★★★	★★★
	JCA	★★★	★★★	★★★	★★★
	JMX	★★★	★★★	★★★	★★★
	JAAS	★★★	★★★	★★★	★★★
	JNDI	★★★	★★★	★★★	★★★
	JSE	✗	✗	★★★	✗
Langages	Java	★★★	★★★	★★★	★★★
	C / C++	★★	★★★	★	✗
	JavaScript	✗	★★★	✗	✗
	.Net	✗	★★★	✗	✗
	Delphi	✗	★★★	✗	✗

		JORAM	AMQ	OMQ	JMQ
	Perl	✗	★★★	✗	✗
	PHP	✗	★★★	✗	✗
	Pike	✗	★★★	✗	✗
	Python	✗	★★★	✗	✗
	Ruby	✗	★★★	✗	✗
Gestion des Messages	Hiérarchie de Topic	★★	★	✗	✗
	Priorité	★★	✗	✗	✗
	WildCard	✗	★★	✗	★★
	Groupe de Messages	✗	★★★	✗	✗
	Destination Virtuelle	✗	★★★	✗	✗
	EIP	✗	★★★	✗	✗
Persistance	Système de Fichier (Normal / Optimisé)	★★	★★★	★★	★★
	Compatible JDBC	★★	★★★	★★	★★
Topologie	Configuration	★★	★★	★★★	★★
	Multi-site	★★★	★★	★★	★★
	Réplication	★★	★★	✗	★★
	Découverte par Multicast	✗	★★★	✗	✗
	Découverte par Broadcast	✗	★★★	✗	✗
	Découverte par LDAP	✗	★★★	✗	✗
Intégration	E.J.B	★★★	★★★	✗	★★★
	Spring	★★★	★★★	✗	✗
	Standalone	★★	★★★	★★	★
Configuration	Fichier (XML / Non standard/N)	★★	★★★	★	★★
	À la volé et à chaud	★	★★	★★★	✗

		JORAM	AMQ	OMQ	JMQ
Administration / Monitoring	Par Messages	✗	★★	★★	✗
	Interface graphique fournie	★	★★★	★★★	✗
	Interface script fournie	✗	✗	★★★	✗
Sécurité	SSL / TLS	★★★	★★★	★★★	★★★
	Gestion Utilisateur	★★★	★★★	★★	★★
	Gestion de Groupe	✗	★★★	★★★	★★★
	Gestion des droits par domaine	★★★	★★★	★★★	★★★
	JAAS	★★★	★★★	★★★	★★★
Autre MOM	Passerelle JMS fournie	★★	★★★	★★	✗
Divers	Version	5.2	5.2	4.2	1.4.4
	Nombre de contributeur	24	114	?	4
	Nombre de pages du site	73	14200	62	17
	Page Rank du site	4	8	6	6
	Licence	LGPL	Apache 2	CDDL ou GPL v2	LGPL

BENCHMARK DE DÉBIT

Scénario de test

Ce test de performance a pour but de mettre en exergue les limites des MOMs selon la charge infligée. Pour ce faire, nous allons mettre en place un MOM et lui envoyer des messages à débit constant et pendant une période de 10 secondes. Nous allons mesurer pour chaque message le temps écoulé de l'envoi, jusqu'à sa réception.

Le test est organisé en phases de durées égales. Chaque phase se caractérise par un débit constant. Pour chaque débit, nous obtenons plusieurs valeurs. Afin d'éviter toute interférence, chaque phase est séparée d'une autre par une vidange du MOM. Cette vidange se fait naturellement en attendant que tous les messages soient consommés.

Nous répétons ce test avec trois tailles de messages différentes : 0.1, 1 et 10 Ko.

Réalisation du test

Après la mise en place d'un MOM, nous lançons les programmes « producteur » et « consommateur ».

Le programme « producteur » agrège plusieurs producteurs JMS. En effet, afin de soutenir un débit constant, le programme se divise en multiples *threads*. On obtient ainsi un producteur par seconde délivrant N messages par intervalle de temps. Le nombre N correspond donc au débit souhaité. Chaque message est daté et identifié. Afin de vraiment soutenir le débit souhaité, chaque producteur ne peut envoyer que 1000 Messages.

Le programme « consommateur », quant à lui, ne reçoit pas nécessairement les messages dans le bon ordre. Il les récolte, les horodates, les analyses et les regroupe par seconde. À chaque fin de phase, il produit ses résultats. L'analyse des messages se fait à la fin afin de ne pas perturber le test.

Configuration

La configuration des deux outils est issue de celle par défaut. Elle est épurée de tout ce qui n'est pas nécessaire. Le mode de transport est le TCP. Aucun chiffrement particulier n'a été mis en place et aucune limite de mémoire au niveau de la configuration non plus. Au niveau de la JVM, 7 Go lui ont été alloués pour chaque broker, consommateur et producteur.

La machine

Les producteurs, le consommateur et le broker tournent sur des machines EC2 distinctes, allouées sur le *cloud* Amazon, du type :

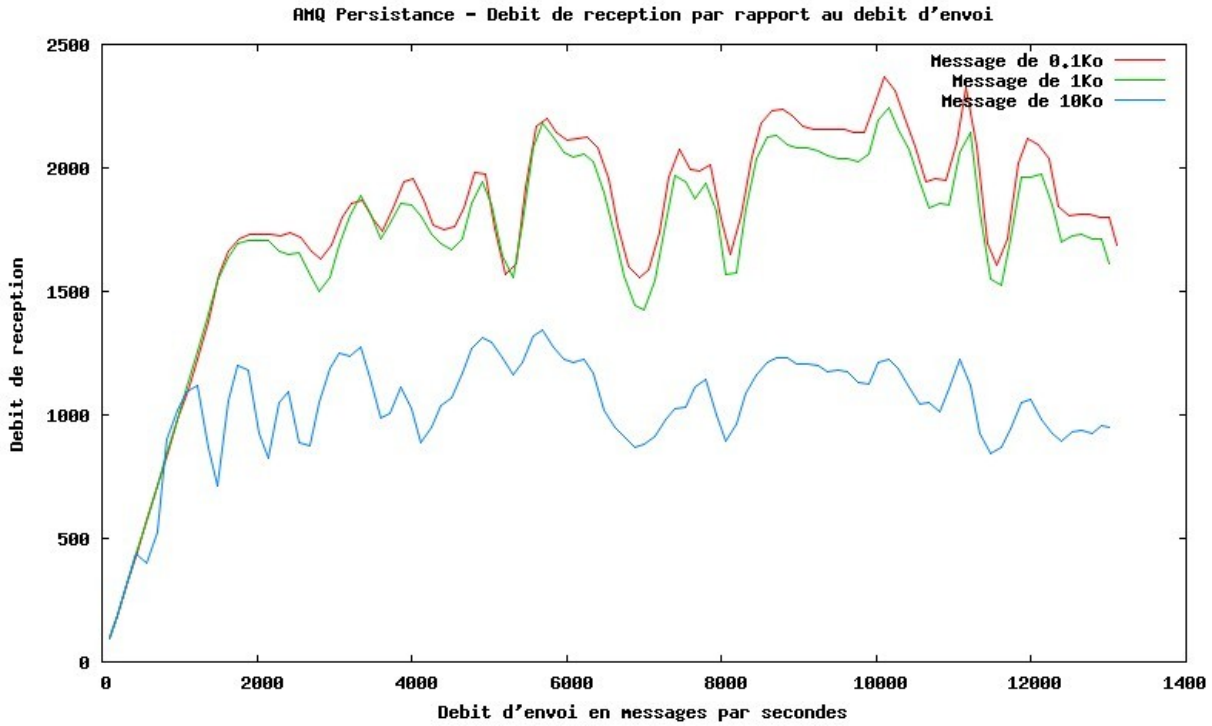
- 4 unités de traitement 64 bits
- 7.5 Go de R.A.M
- 850 Go de Disque dur

Chaque unité de traitement est équivalente à 1.0-1.2 GHz Opteron 2007 ou à Xeon 2007.

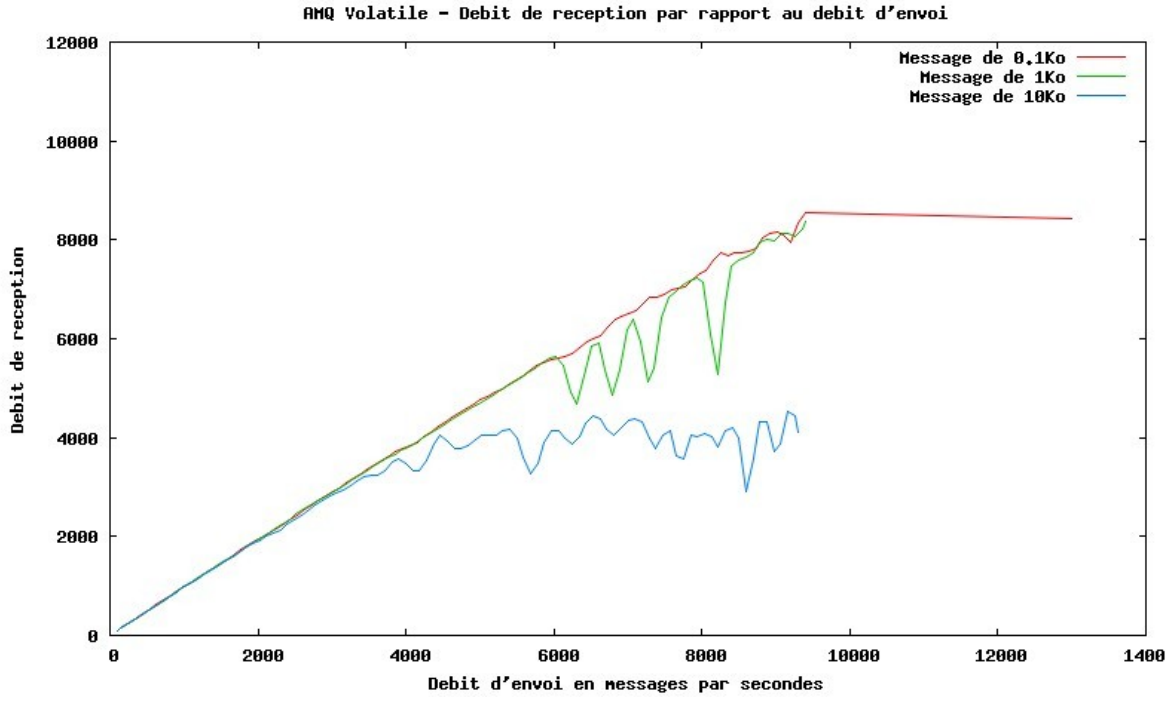
Résultats du test

Nous allons exprimer chaque résultat selon le débit de réception par rapport au débit d'envoi.

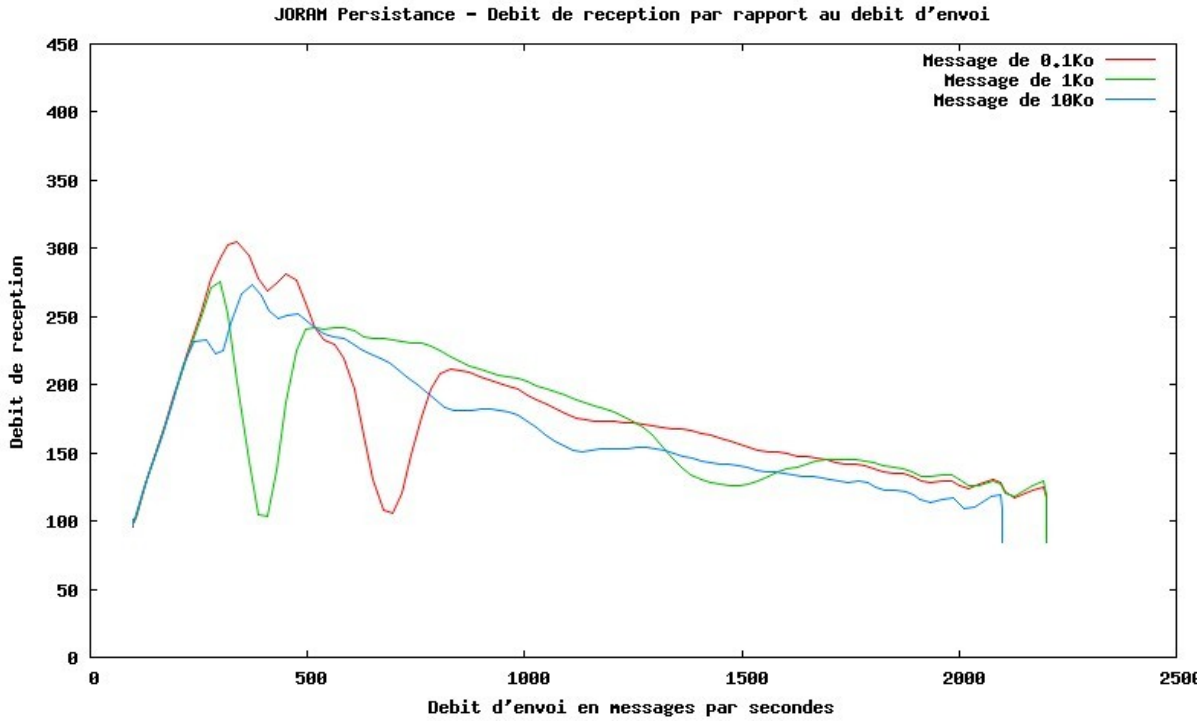
Active MQ avec Persistance



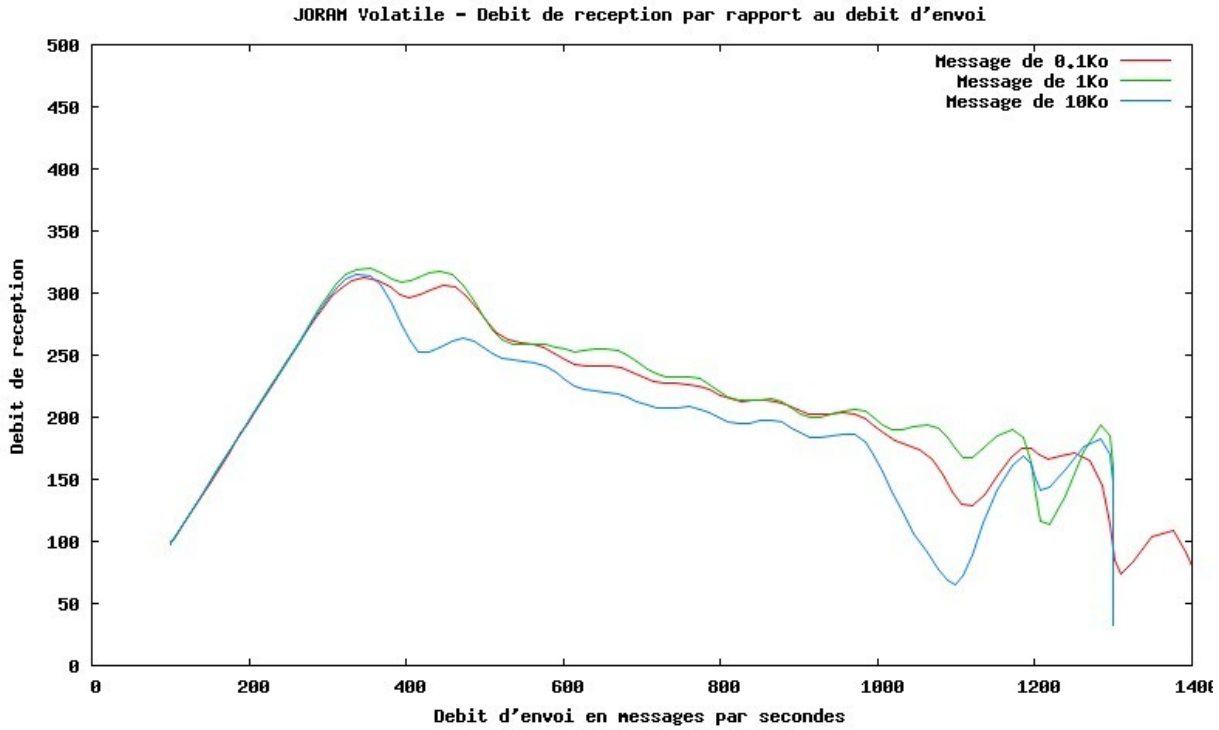
Active MQ, sans Persistance (volatile)



Joram avec Persistance



JORAM sans Persistance (volatile)



Analyse

On remarque que les deux outils ne réagissent vraiment pas de la même manière.

D'une part, le débit de réception d'active MQ se stabilise tandis que le débit de réception de Joram chute progressivement au fur et à mesure que le débit en entrée augmente.

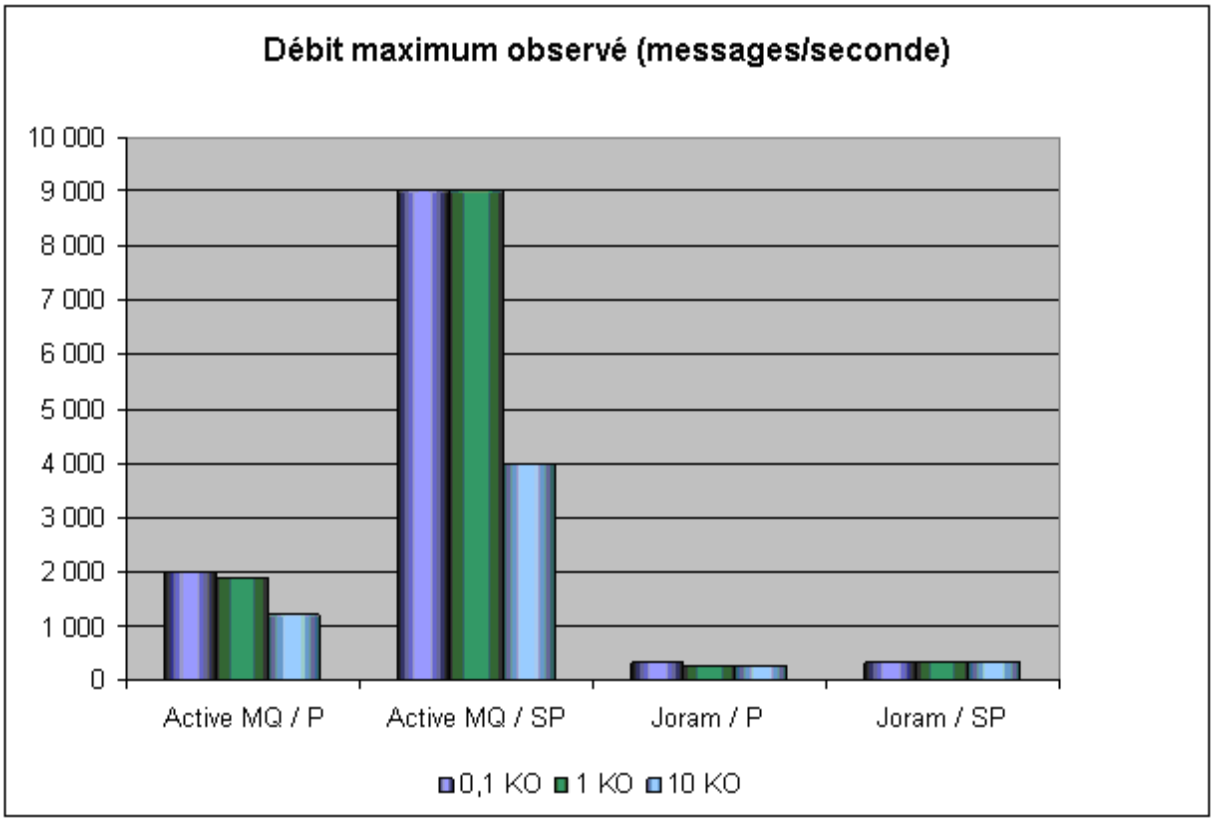
Active MQ supporte mieux la charge que JORAM.

D'autre part, JORAM n'est pas aussi sensible que Active MQ à la taille des messages. On remarque que la différence entre les débits de réception des messages de différentes tailles est plus grande dans le cas d'Active MQ que celle de JORAM.

Une chose est sûre, Active MQ est bien plus performant que JORAM, à petite ou forte charge. Voici un tableau récapitulatif des débits de réception.

Taille des messages		**Débit possible, en messages par seconde**			
		ACTIVE MQ		**JORAM**	
		Max	Moyenne	Max	Moyenne
0.1 Ko	P	2 400	2 000	320	n/a
	SP	9 000	9 000	330	n/a
1 Ko	P	2 350	1 900	270	n/a
	SP	9 000	9 000	320	n/a
10 Ko	P	1 300	1 200	270	n/a
	SP	4 000	4 000	320	n/a

Notons que « P » signifie ici « avec Persistance », et « SP » signifie « Sans Persistance ».



On remarque aussi que la différence entre persistance et sans persistance est très grande pour Active MQ. Le débit de réception varie avec un facteur de 3.

Au final, on retiendra que, dans un mode sans persistance, Active MQ achemine jusqu'à 9000 messages par seconde, et jusqu'à 2000 avec persistance.

SYNTHÈSE

La première question n'est pas quelle solution de MOM choisir. L'important est d'abord de bien identifier les bénéfices importants qu'un MOM peut apporter dans un système d'information, et c'est pourquoi nous nous sommes attachés en premier lieu de bien décrire les services rendus par un MOM, et la manière dont il pouvait simplifier et fiabiliser les interactions entre applications.

Les MOMs sont encore trop peu connus des architectes, et on voit souvent mettre en œuvre des échanges FTP, ou bien des appels synchrones trop fragiles, ou autres moyens d'échanges rudimentaires, voir archaïques. Les MOMs apportent une solution ouverte, flexible et extensible à une diversité de problèmes d'intégration. On peut déployer un MOM dans un contexte hautement hétérogène, mais il a toute sa place également au sein d'une simple plateforme web, un peu haut de gamme.

Une fois que l'architecte est convaincu qu'un middleware de type MOM est le bon socle d'échange pour sa plateforme, il lui reste à faire le choix d'un produit. L'offre est riche, et comme on l'a vu, tous les produits convergent autour de la spécification JMS, ce qui offre un niveau de service de base commun, mais aussi permet de concentrer l'expertise.

Lorsque nous faisons, dans nos livres blancs, un panorama des solutions open source du marché, il arrive souvent que l'on ne puisse conclure à la supériorité claire d'un produit en particulier. La conclusion est alors que selon les besoins spécifiques d'un projet, selon le contexte d'insertion, tel ou tel produit arrivera en tête.

Mais sur le sujet des MOMs, force est de constater qu'un produit sort du lot : notre étude nous amène à conclure que Apache Active MQ est la meilleure des quatre solutions étudiées :

- Elle a la pérennité et la légitimité de la fondation Apache, s'appuie sur un socle de produits de qualité de la fondation, et semble faire converger une communauté de développement plus large et active.

- Elle offre une couverture fonctionnelle plus large, sur à peu près tous les plans, avec en particulier l'intégration possible de traitements et d'aiguillages.

- Elle est particulièrement extensible, et peu satisfaire aussi bien des besoins simples que de vastes problématiques d'entreprise.

- Et enfin, elle présente des performances supérieures.

Pour nous, l'affaire est entendue, Active MQ nous semble être le meilleur choix. Sauf bien sûr si l'on a par ailleurs déjà déployé une infrastructure basée sur les autres lignes de produits : Redhat/JBoss, SUN/GlassFish, ou OW2/Jonas.

La question ensuite, sera d'ajuster l'ambition qui est donnée au middleware dans son infrastructure applicative. L'avantage des MOMs est leur relative simplicité : ils ne prétendent pas tout faire, mais ce qu'ils font ils le font de manière fiable et performante. La limitation essentielle des MOMs, comme on l'a vu, est qu'ils ne s'occupent pas du contenu du message, et supposent donc que les applications *parlent le même langage, se sont entendues sur un format commun*. Dans un environnement hétérogène, incluant du patrimoine ancien, on voit bien qu'on ne pourra faire cette hypothèse.

Active MQ, avec l'intégration de Apache Camel, prend des aspects d'EAI, et peut prendre en charge des transformations de messages et conversions de formats, mais de manière encore relativement limitée.

Pour prendre en charge une plus forte hétérogénéité, et s'ouvrir vers un plus large éventail de modes de connexions et de protocoles, il faudra considérer un ESB, *Enterprise Service Bus*, base d'une architecture SOA globale. Le principal ESB dans le monde de l'open source est MULE, de MuleSource, que nous apprécions particulièrement chez Smile.

Mais en matière d'architecture, il est essentiel de savoir ajuster l'ambition au problème, et les solutions les plus sophistiquées ne sont pas toujours les plus appropriées.

Les simples MOM, et Active MQ en particulier, restent donc des produits extrêmement pertinents et utiles pour construire des plateformes distribuées, ou permettre l'interopérabilité d'un petit nombre d'applications.

Depuis plusieurs années, Smile a construit une expertise des middleware au service d'architectures extensibles et performantes, et nos experts seront heureux de vous aider à tirer le meilleur parti d'une solution MOM open source.

Les livres blancs Smile

Les livres blancs Smile sont téléchargeables gratuitement sur www.smile.fr

- **Introduction à l'open source et au Logiciel Libre**

  Son histoire, sa philosophie, ses grandes figures, son marché, ses modèles économiques, ses modèles de support et modèles de développement. [52 pages]

- **Gestion de contenus : les solutions open source**

  Dans la gestion de contenus, les meilleures solutions sont open source. Du simple site à la solution entreprise, découvrez l'offre des CMS open source. [58 pages]

- **Portails : les solutions open source**

  Pour les portails aussi, l'open source est riche en solutions solides et complètes. Après les CMS, Smile vous propose une étude complète des meilleures solutions portails. [50 pages]

- **200 questions pour choisir un CMS**

  Toutes les questions qu'il faut se poser pour choisir l'outil de gestion de contenu répondra le mieux à vos besoins. [46 pages]

- **Conception d'applications web**

  Synthèse des bonnes pratiques pour l'utilisabilité et l'efficacité des applications métier construites en technologie web. [61 pages]

- **Les frameworks PHP**

  Une présentation complète des frameworks et composants qui permettent de réduire les temps de développement des applications, tout en améliorant leur qualité. [77 pages]

- **Les 100 bonnes pratiques du web**

  Cent et quelques « bonnes pratiques du web », usages et astuces, incontournables ou tout simplement utiles et qui vous aideront à construire un site de qualité. [26 pages]

- **ERP/PGI: les solutions open source**

  Des solutions open source en matière d'ERP sont tout à fait matures et gagnent des parts de marché dans les entreprises, apportant flexibilité et coûts réduits. [121 pages]

- **GED : les solutions open source**

  Les vraies solutions de GED sont des outils tout à fait spécifiques ; l'open source représente une alternative solide, une large couverture fonctionnelle et une forte dynamique. [77 pages]

- **Référencement : ce qu'il faut savoir**

  Grâce à ce livre blanc, découvrez comment optimiser la "référençabilité" et le positionnement de votre site lors de sa conception. [45 pages]

- **Décisionnel : les solutions open source**

  Découvrez les meilleurs outils et suites de la business intelligence open source. [78 pages]

- **Collection Système et Infrastructure :**
  - Virtualisation open source [41 pages]
  - Architectures Web open source [177 pages]
  - Firewalls open source [58 pages]
  - VPN open source [31 pages]
  - Cloud Computing [42 pages]
  - Middleware [91 pages]

Smile
OPEN SOURCE SOLUTIONS

Contactez-nous, nous serons heureux de vous présenter nos réalisations de manière plus approfondie !
+33 1 41 40 11 00 / sdc@smile.fr