Annexe E

Joram : un intergiciel de communication asynchrone

Ce chapitre décrit les principes de conception et de réalisation de JORAM (*Java Open Reliable Asynchronous Messaging*), un intergiciel de communication asynchrone qui met en œuvre la spécification JMS (*Java Messaging Service*) pour la communication par messages entre applications Java.

La section 1 est un bref rappel sur les intergiciels à messages. La section 2 décrit les traits principaux du modèle de programmation défini par JMS. La section 3 présente les principes directeurs de conception de l'architecture interne de JORAM pour répondre aux objectifs d'une mise en œuvre distribuée et configurable. La section 4 décrit quelques fonctions avancées du système JORAM, pour la répartition de charge, la haute disponibilité et l'interopérabilité avec le monde extérieur. La section 5 présente rapidement le système d'agents distribué qui constitue la base technologique pour la mise en œuvre de JORAM. La section 6 conclut sur les utilisations de JORAM et ses perspectives d'évolution.

E.1 Introduction

Dans cette section, nous présentons les principales caractéristiques des systèmes à messages et les motivations pour leur usage, avant une brève introduction à JMS et à Joram, sujets qui seront développés dans les sections suivantes.

E.1.1 Les intergiciels à messages (MOM)

Les systèmes de communication asynchrones, fondés sur l'envoi de messages, connaissent aujourd'hui un regain d'intérêt dans le contexte des applications réparties sur Internet. En effet, on s'accorde à penser que les modèles de communication asynchrones sont mieux adaptés que les modèles synchrones (de type client-serveur) pour gérer les interactions entre systèmes faiblement couplés. Le couplage faible résulte de plusieurs facteurs de nature spatiale ou temporelle : l'éloignement géographique des entités communicantes, la possibilité de déconnexion temporaire d'un partenaire en raison d'une panne ou d'une interruption de la communication (pour les usages mobiles par exemple). Les modèles de communication asynchrones prennent en compte l'indépendance entre entités communicantes et sont donc mieux armés pour traiter ces problèmes. Aujourd'hui les systèmes de communication asynchrones, appelés MOM (*Message Oriented Middleware*), sont très largement répandus dans la mesure où ils constituent la base technologique pour la réalisation des classes d'applications suivantes :

- Intégration de données et intégration d'applications (EAI, B2B, ESB, etc.)
- Systèmes ubiquitaires et usages de la mobilité.
- Surveillance et contrôle d'équipements en réseau.

Du point de vue du modèle de communication les intergiciels à messages partagent les concepts suivants :

- Les entités communicantes sont découplées. L'émission (on dit aussi la production) d'un message est une opération non bloquante. D'autre part émetteur (producteur) et récepteur (consommateur) ne communiquent pas directement entre eux mais utilisent un objet de communication intermédiaire (boîte aux lettres).
- Deux modèles de communication sont fournis : un modèle point à point dans lequel un message produit est consommé par un destinataire unique ; un modèle multipoints dans lequel un message peut être adressé à une communauté de destinataires (communication de groupe).
- Le mode multipoints est généralement doublé d'un système de désignation associative dans lequel les destinataires du message sont identifiés par une propriété du message. Ce mode de communication est à l'origine du modèle Publication/Abonnement (*Publish/Subscribe*) largement répandu aujourd'hui.

Pendant de nombreuses années l'essor des systèmes de communication asynchrones a été retardé par l'absence de normalisation (au contraire de ce qui s'est passé pour les systèmes client-serveur avec CORBA). Les intergiciels à messages sont donc restés propriétaires à la fois du point de vue du modèle de programmation et de leur implémentation. A la fin des années 90, l'émergence de la spécification JMS (*Java Messaging Service*) dans le monde Java a partiellement comblé ce handicap, partiellement seulement car, comme nous le verrons plus loin, la norme définit le modèle de programmation et l'API correspondante. En revanche il n'y a toujours pas de tentative de normalisation sur l'intergiciel lui-même, ni même sur les mécanismes d'interopérabilité comme c'est le cas dans CORBA avec le protocole IIOP.

E.1.2 JMS (*Java Messaging Service*)

JMS (*Java Messaging Service*) est la spécification d'un service de messagerie en Java. Plus précisément JMS décrit l'interface de programmation pour utiliser un bus à messages asynchrone, c'est-à-dire la manière de programmer les échanges entre deux composants applicatifs.

Dans la structure d'une application JMS, on distingue les composants suivants (voir Figure 1).

- Le système de messagerie JMS (*JMS provider*). Dans la mise en oeuvre du système de messagerie on distingue le service de base qui met en œuvre les abstractions du

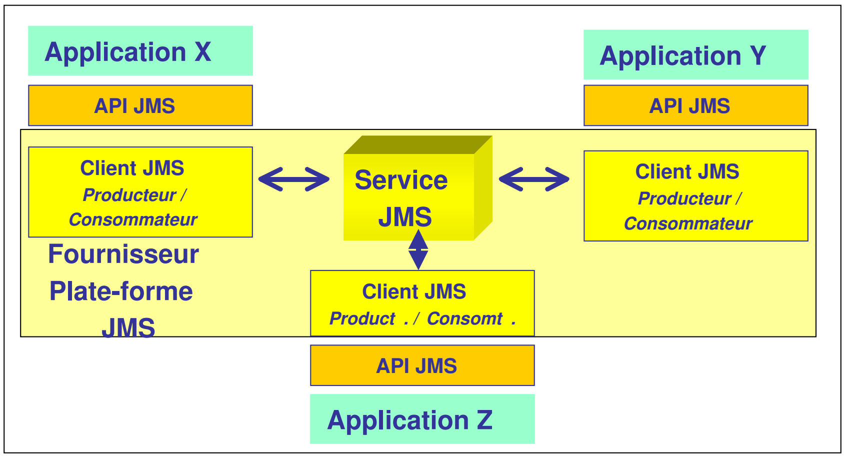

Figure E.1 – Application JMS

modèle de programmation JMS et une bibliothèque de fonctions liée aux programmes utilisateurs qui met en œuvre l'interface de programmation JMS.

- Les clients JMS (*JMS client*) sont les programmes (applications), écrits en langage Java, qui produisent (émettent) et consomment (reçoivent) des messages selon des protocoles spécifiés par l'API JMS.
- Les messages JMS sont des objets qui permettent de véhiculer des informations entre des clients JMS. Différents types de messages sont utilisables : texte structuré (par exemple un fichier XML), format binaire, objets Java (sérialisés), etc.

Deux modes de communication (*Messaging Domains*) sont fournis par la spécification JMS : La communication point à point est fondée sur des files de message (`Queues`). Un message est adressé à une queue par un client producteur (flot noté 1) d'où il est extrait par un client consommateur. Un message est consommé par un seul client. Le message est stocké dans la queue jusqu'à sa consommation ou jusqu'à l'expiration d'un délai d'existence. La consommation d'un message peut être synchrone (retrait explicite par le consommateur – flots 2 et 3) ou asynchrone (activation d'une procédure de veille préenregistrée chez le consommateur - non représenté dans la figure 2) par le gestionnaire de messages. La consommation du message est confirmée par un accusé de réception généré par le système ou le client.

La communication multipoints est fondée sur le modèle publication/abonnement (*Publish/Subscribe*) - voir figure 3. Un client producteur émet un message concernant un sujet prédéterminé (`Topic`) (flot noté 2 dans la figure 3). Tous les clients préalablement abonnés à ce `Topic` (flots 1a et 1b) reçoivent le message correspondant (flots 3a et 3b). Le modèle de consommation (implicite/explicite) est identique à celui du mode point à point.

La dernière version de la spécification JMS (dénotée JMS 1.1) unifie la manipulation des deux modes de communication au niveau du client JMS en introduisant la notion de `Destination`[1] pour représenter soit une queue de message, soit un `Topic` . Cette unification simplifie l'API (les primitives de production/consommation sont syntaxiquement fusionnées) et permet une optimisation des ressources de communication. Elle permet

[1] Dans la suite ce terme « destination » sera utilisé pour faire référence indistinctement à une queue ou à un `Topic` .

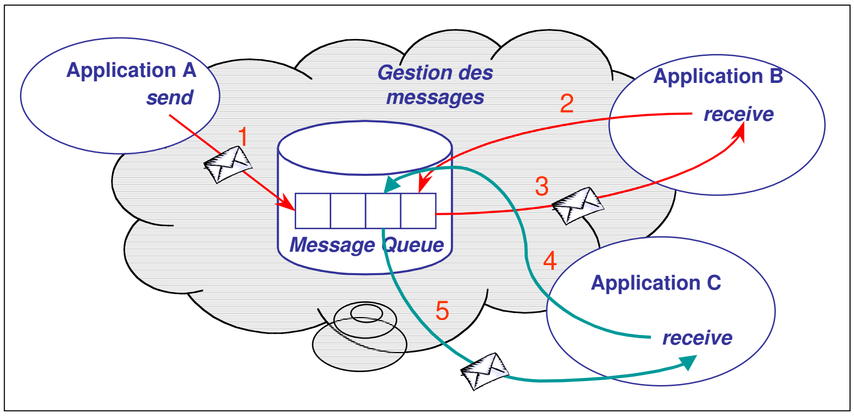

Figure E.2 – Modèle de communication Point à Point

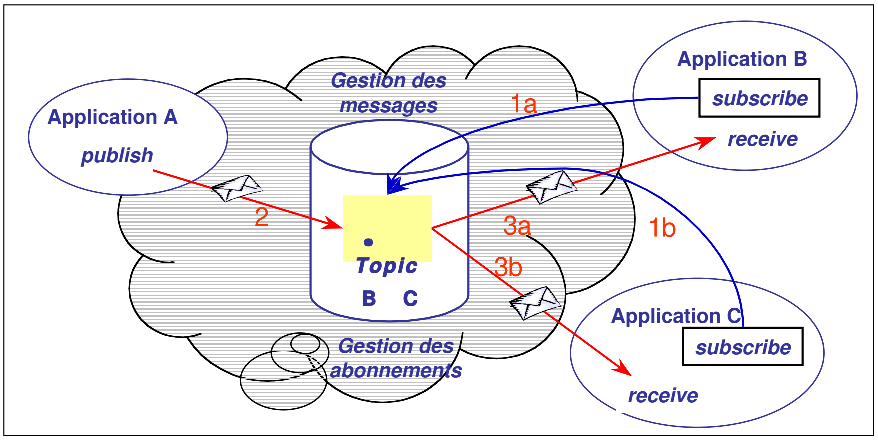

Figure E.3 – Modèle de communication *Publish/Subscribe* également d'utiliser le mode transactionnel dans les deux modèles de communication. Bien entendu cela ne remet pas en cause la sémantique propre à chacun de ces modes de communication, qui doit être prise en compte dans la programmation du client JMS. Enfin, la spécification JMS définit des options de qualité de service, en particulier pour ce qui concerne les abonnements (temporaires ou durables) et la garantie de délivrance des messages (propriété de persistance).

E.1.3 Ce que JMS ne dit pas et ne fait pas

JMS définit un protocole d'échange entre des producteurs et des consommateurs de messages. JMS ne donne aucune indication sur la mise en œuvre du service de messagerie qui est donc une implantation propriétaire chez tous les fournisseurs de plates-formes JMS. JMS définissant l'API d'accès au service de messagerie, une application JMS est supposée être indépendante d'une plate-forme JMS. Cette indépendance répond à un objectif de portabilité. Dans la réalité la portabilité est réduite du fait que les fonctions d'administration sont propres à chaque plate-forme. Par ailleurs l'interopérabilité entre deux clients JMS implantés sur des plates-formes différentes n'est pas garantie car elle requiert l'existence d'une passerelle spécifique pour mettre en correspondance les mécanismes des deux plates-formes. Cette fonction est disponible dans certaines offres du marché pour un nombre restreint de plates-formes. Joram fournit également cette fonction (voir 4.3). La spécification JMS n'est pas complète. Certaines fonctions essentielles au fonctionnement d'une plate-forme JMS ne sont pas décrites dans la spécification et font donc l'objet d'implantations propriétaires. 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. À l'opposé, la plupart des plates-formes proposent des fonctions additionnelles qui se présentent comme des valeurs ajoutées aux offres concurrentes (par exemple les `Topics` hiérarchisés, des fonctions de sécurité et des mécanismes de haute disponibilité, etc.). Sur ces deux points Joram n'échappe pas à la règle. En ce qui concerne l'administration, la spécification JMS est limitée à la définition d'un certain nombre d'objets d'administration qui sont gérés par le service de messagerie et qui sont utilisés par les clients JMS au travers de fonctions spécifiques de l'API JMS.

E.1.4 Joram : une mise en œuvre d'un service de messagerie JMS

Joram (*Java Open Reliable Asynchronous Messaging*) est une implémentation 100% Java de la spécification JMS. Joram est conforme aux dernières spécifications JMS 1.1 (elles-mêmes étant partie intégrante de la spécification J2EE 1.4). Joram est un composant *open source* disponible sur la base de code ObjectWeb sous licence LGPL - http ://joram.objectweb.org . Joram est aujourd'hui en exploitation dans de nombreux environnements opérationnels où il est utilisé de deux façons complémentaires :

- Comme un système de messagerie Java autonome entre des applications JMS développées pour des environnements variés (de J2EE à J2ME).
- Comme un composant de messagerie intégré dans un serveur d'application J2EE. A ce titre, Joram est une brique essentielle du serveur J2EE JOnAS, également disponible sur ObjectWeb - http ://jonas.objectweb.org

Comme toutes les autres réalisations de plates-formes JMS, JORAM est structuré en deux parties : une partie «serveur JORAM» qui gère les objets JMS (`Queues`, `Topics` , connexions, etc.) et une partie « client JORAM » qui est liée à l'application cliente JMS. Comme nous le verrons plus loin en détaillant l'architecture de JORAM, le serveur JORAM peut être mis en œuvre de façon centralisée ou de façon distribuée. La communication entre un client JORAM et un serveur JORAM s'appuie sur le protocole TCP/IP. Une variante, présentée plus loin, consiste à utiliser le protocole HTTP/SOAP pour les clients JMS développés dans un environnement J2ME. La communication entre deux serveurs JORAM peut utiliser différents types de protocoles selon les besoins (TCP/IP, HTTP, SOAP, communication sécurisée via SSL). Client et serveurs peuvent être sur des machines physiques différentes ; ils peuvent partager la même machine et s'exécuter dans des processus différents ou partager le même processus.

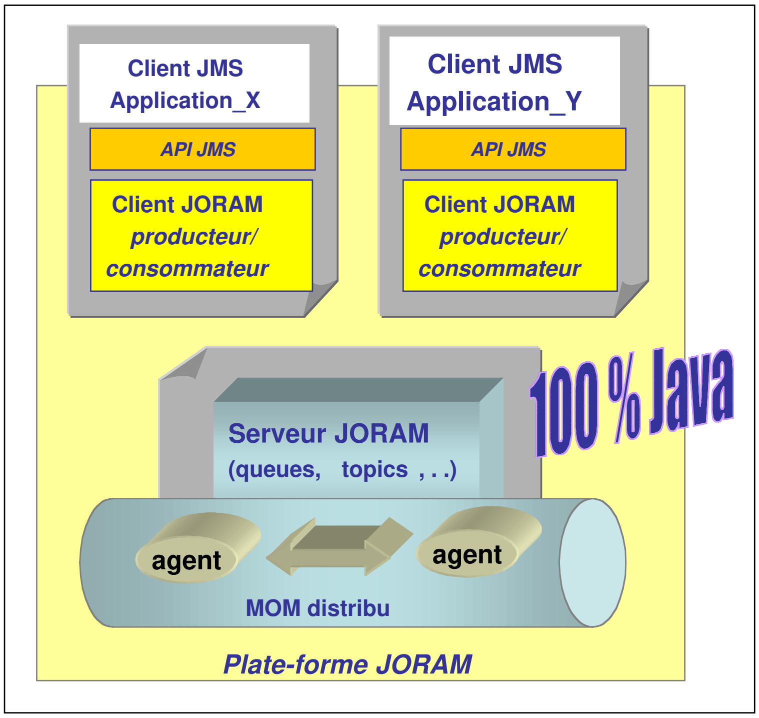

Figure E.4 – La plate-forme JORAM

En résumé, JORAM est une plate-forme JMS disponible en *open source*, dont la réalisation s'appuie sur un intergiciel à messages qui exploite une technologie d'agents distribués, présentée brièvement en section 4. La structure générale de la plate-forme JORAM est représentée sur la figure 4.

E.2 Le modèle de programmation JMS

Cette section présente les principes directeurs du modèle de programmation JMS. La description détaillée de l'API JMS n'est pas un objectif de ce papier (à cet effet le lecteur pourra se référer au tutoriel disponible en ligne). Nous exposons ici uniquement les éléments nécessaires à la compréhension des aspects architecturaux développés plus loin.

E.2.1 Les abstractions de JMS

JMS définit un ensemble de concepts communs aux deux modèles de communication Point-à-Point et *Publish/Subscribe* :

- `ConnectionFactory` : un objet d'administration utilisé par le client JMS pour créer une connexion avec le système de messagerie.
- `Connection` : une connexion active avec le système de messagerie.
- `Destination` : l'objet de communication entre deux clients JMS. Cet objet désigne la destination des messages pour un producteur et la source des messages attendus pour un consommateur. La destination désigne une `Queue` de messages dans le modèle de communication point à point et un `Topic` dans le modèle de communication *Publish/Subscribe*.
- `Session` : un contexte (mono-thread) pour émettre et recevoir des messages.
- `MessageProducer` : un objet créé par une session et utilisé pour émettre des messages à un objet `Destination`.
- `MessageConsumer` : un objet créé par une session et utilisé pour recevoir les messages déposés dans un objet `Destination`.

Ces objets se déclinent selon le modèle de communication choisi, comme le montre le tableau de la figure 5.

Interface ˙ parent ¨	*Point- -point*	*Publish/Subscribe*
Destination	**Queue**	**Topic**
ConnectionFactory	**QueueConnectionFactory**	**TopicConnectionFactory**
Connection	**QueueConnection**	**TopicConnection**
Session	**QueueSession**	**TopicSession**
MessageProducer	**QueueSender**	**TopicPublisher**
MessageConsumer	**QueueReceiver**	**TopicSubscriber**

Figure E.5 – Correspondance des interfaces *Point à Point* et *Publish/Subscribe*

E.2.2 Principe de fonctionnement

Un client JMS exécute la séquence d'opérations suivante :

- Il recherche un objet `ConnectionFactory` dans un répertoire en utilisant l'API JNDI (*Java Naming and Directory Interface*)
- Il utilise l'objet `ConnectionFactory` pour créer une connexion JMS, il obtient alors un objet `Connection`.
- Il utilise l'objet `Connection` pour créer une ou plusieurs sessions JMS, il obtient alors des objets `Session`.
- Il utilise le répertoire pour trouver un ou plusieurs objets `Destination`.
- Il peut alors, à partir d'un objet `Session` et des objets `Destination`, créer les objets `MessageProducer` et `MessageConsumer` pour émettre et recevoir des messages.

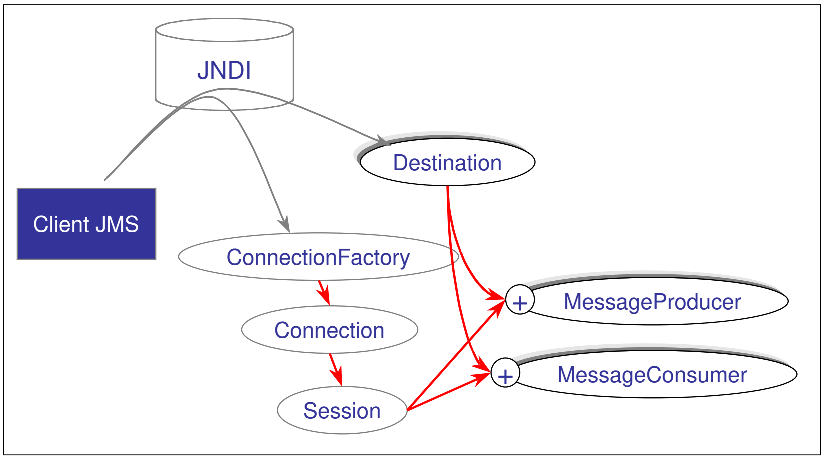

Figure E.6 – Architecture d'une application JMS

Les étapes et les objets caractérisant une application JMS sont résumés dans la figure 6.

E.2.3 Messages

Les systèmes de messagerie manipulent les messages comme des entités comportant un entête et un corps. L'en-tête contient l'information utile pour l'identification et le routage du message ; le corps contient les données utiles à l'application. Le modèle de message JMS répond aux critères suivants :

- Il fournit une interface de message unique.
- Il supporte les messages contenant du texte, des objets Java ou des données XML.

Structure des messages

Un message JMS est composé de trois parties :

- Un en-tête : qui contient l'ensemble des données utilisées à la fois par le client et le système pour l'identification et l'acheminement du message.
- Des propriétés : en plus des champs standard de l'en-tête, les messages permettent d'ajouter des champs optionnels sous forme de propriétés normalisées, ou de propriétés spécifiques à l'application ou au système de messagerie.
- Un corps : JMS définit différents types de corps de message afin de répondre à l'hétérogénéité des systèmes de messagerie.

En-tête des messages

Un en-tête de message JMS est composé des champs suivants :

- `JMSDestination` : ce champs contient la destination du message, il est fixé par la méthode d'envoi de message en fonction de l'objet `Destination` spécifié.
- `JMSDeliveryMode` : ce champ définit le mode de délivrance du message (persistant

ou non[2]) ; il est fixé par la méthode d'envoi de message en fonction des paramètres spécifiés.
- `JMSMessageId` : ce champs contient un identificateur qui identifie de manière unique chaque message envoyé par un système de messagerie. Il est fixé par la méthode d'envoi de message et peut-être consulté après envoi par l'émetteur.
- `JMSTimeStamp` : ce champs contient l'heure de prise en compte du message par le système de messagerie, il est fixé par la méthode d'envoi de message.
- `JMSReplyTo` : ce champs contient la `Destination` à laquelle le client peut éventuellement émettre une réponse. Il est fixé par le client dans le message.
- `JMSExpiration` : ce champs est calculé comme la somme de l'heure courante (GMT) et de la durée de vie d'un message (time-to-live). Lorsqu'un message n'est pas délivrée avant sa date d'expiration il est détruit ; aucune notification n'est définie pour prévenir de l'expiration d'un message.
- `JMSCorrelationId`, `JMSPriority`, `JMSRedelivered`, `JMSType`, etc.

Propriétés

Les propriétés permettent à un client JMS de sélectionner les messages en fonction de critères applicatifs. Un nom de propriété doit être de type `String` ; une valeur peut être : `null, boolean, byte, short, int, long, float, double` et `String`. Un client peut ainsi définir des filtres en réception dans l'objet `MessageConsumer` à l'aide d'une chaîne de caractère dont la syntaxe est basée sur un sous-ensemble de la syntaxe d'expression de conditions du langage SQL.

Corps du message

JMS définit cinq formes de corps de messages :
- `StreamMessage` : un message dont le corps contient un flot de valeurs de types primitifs Java ; il est rempli et lu séquentiellement.
- `MapMessage` : un message dont le corps est composé d'un ensemble de couples noms-valeurs.
- `TextMessage` : un message dont le corps est une chaîne de caractère (`String`).
- `ObjectMessage` : un message dont le corps contient un objet Java sérialisable.
- `BytesMessage` : un message dont le corps est composé d'un flot d'octets ; ce type de message permet de coder un message conformément à une application existante.

E.2.4 Objets JMS

Objets d'administration

JMS ne définit pas une syntaxe d'adressage standard. En lieu et place, il définit l'objet `Destination` qui encapsule les différents formats d'adresses des systèmes de messagerie. L'objet `ConnectionFactory` encapsule l'ensemble des paramètres de configuration définit par l'administrateur, il est utilisé par le client pour initialiser une connexion avec le système de messagerie.

[2]Le mode non persistant correspond à une sémantique au plus une fois, tandis que le mode persistant correspond à une sémantique exactement une fois : le message ne être ni perdu ni dupliqué.

Objet `Connection`

Un objet `Connection` représente une connexion active avec le système de messagerie. Cet objet supporte le parallélisme, c'est-à-dire qu'il fournit plusieurs voies de communication logiques entre le client et le serveur. Lors de sa création, le client peut devoir s'authentifier. L'objet `Connection` permet de créer une ou plusieurs sessions. Lors de sa création une connexion est dans l'état stoppé[3]. Elle doit être explicitement démarrée pour recevoir des messages. Une connexion peut être temporairement stoppée puis redémarrée ; après son arrêt, il peut y avoir encore quelques messages délivrés ; arrêter une connexion n'affecte pas sa capacité à transmettre des messages. Du fait qu'une connexion nécessite l'allocation de nombreuses ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utile.

Objet `Session`

Un objet JMS `Session` est un contexte *mono-thread* pour produire et consommer des messages. Il répond à plusieurs besoins :

- Il construit les objets `MessageProducer` et `MessageConsumer`.
- Il crée les objets `Destination` et `Message`.
- Il supporte les transactions et permet de grouper plusieurs opérations de réception et d'émission dans une unité atomique qui peut être validée (resp. invalidée) par la méthode `Commit` (resp. `Rollback`).
- Il réalise l'ordonnancement des messages reçus et envoyés.
- Il gère l'acquittement des messages.

Bien qu'une session permette la création de multiples objets `MessageProducer` et `MessageConsumer`, ils ne doivent être utilisés que par un flot d'exécution à la fois (contexte mono-threadé). Pour accroître le parallélisme un client JMS peut créer plusieurs sessions, chaque session étant indépendante. Dans le mode *Publish/Subscribe*, si deux sessions s'abonnent à un même sujet, chaque client abonné (`TopicSubscriber`) reçoit tous les messages émis sur le `Topic` . Du fait qu'une session nécessite l'allocation de ressources dans le système de messagerie, il est recommandé de la fermer lorsqu'elle n'est plus utilisée.

Objet MessageConsumer

Un client utilise un objet `MessageConsumer` pour recevoir les messages envoyés à une destination particulière. Un objet `MessageConsumer` est créé en appelant la méthode `CreateConsumer` de l'objet Session avec un objet `Destination` en paramètre. Un objet `MessageConsumer` peut être créé avec un sélecteur de message pour filtrer les messages à consommer. JMS propose deux modèles de consommation (synchrone et synchrone) pour les messages. Dans le modèle synchrone un client demande le prochain message en utilisant la méthode `Receive` de l'objet `MessageConsumer` (mode de consommation *Pull*). Dans le mode asynchrone il enregistre au préalable un objet qui implémente la classe `MessageListener` dans l'objet `MessageConsumer` ; les messages sont alors délivrés lors de leur arrivée par appel de la méthode `onMessage` sur cet objet (mode de consommation *Push*).

[3] afin de ne pas perturber l'initialisation de l'application par l'arrivée de messages.

Objet MessageProducer

Un client utilise un objet `MessageProducer` pour envoyer des messages à une destination. Un objet `MessageProducer` est créé appelant la méthode `CreateProducer` de l'objet `Session` avec un objet `Destination` en paramètre. Si aucune destination n'est spécifiée un objet `Destination` doit être passé à chaque envoi de message (en paramètre de la méthode `Send`). Un client peut spécifier le mode de délivrance, la priorité et la durée de vie par défaut pour l'ensemble des messages envoyés par un objet `MessageProducer`. Il peut aussi les spécifier pour chaque message.

E.2.5 Communication en mode Point à point

Un objet `Queue` encapsule un nom de file de message du système de messagerie. Rappelons que JMS ne définit pas de fonctions pour créer, administrer ou détruire des queues. Dans les faits, ces fonctions sont réalisées par des outils d'adminsitration propres à chaque plate-forme JMS. Un client utilise un objet `QueueConnectionFactory` pour créer une connexion active (objet `QueueConnection`) avec le système de messagerie. L'objet `QueueConnection` permet au client d'ouvrir une ou plusieurs sessions (objets `QueueSession`) pour envoyer et recevoir des messages. L'interface des objets `QueueSession` fournit les méthodes permettant de créer les objets `QueueReceiver`, `QueueSender` et `QueueBrowser`.

E.2.6 Communication en mode *Publish/Subscribe*

Le modèle JMS *Publish/Subscribe* définit comment un client JMS publie vers et s'abonne à un nœud dans une hiérarchie de sujets. JMS nomme ces nœuds des `Topics`. Un objet `Topic` encapsule un nom de sujet du système de messagerie. JMS ne met aucune restriction sur la sémantique d'un objet `Topic` , il peut s'agir d'une feuille, ou d'une partie plus importante de la hiérarchie[4]. Un client utilise un objet `TopicConnectionFactory` pour créer un objet `TopicConnection` qui représente une connexion active avec le système de messagerie. Un objet `TopicConnection` permet de créer une ou plusieurs sessions (objet `TopicSession`) pour produire et consommer des messages. L'interface des objets `TopicSession` fournit les méthodes permettant de créer les objets `TopicPublisher` et `TopicSubscriber`. Un client utilise un objet `TopicPublisher` pour publier des messages concernant un sujet donné. Il utilise un objet `TopicSubscriber` pour recevoir les messages publiés sur un sujet donné. Les objets `TopicSubscriber` classiques ne reçoivent que les messages publiés pendant qu'ils sont actifs. Les messages filtrés par un sélecteur de message ne seront jamais délivrés ; dans certains cas une connexion peut à la fois émettre et recevoir sur un sujet. L'attribut `NoLocal` sur un objet `TopicSubscriber` permet de ne pas recevoir les messages émis par sa propre connexion.

E.2.7 JMS par l'exemple

Cette section présente quelques exemples de séquences de code associées aux principales opérations de l'API JMS.

[4]Pour s'abonner à une large classe de sujets par exemple.

Initialisation d'une session "Point-to-Point"

```
// We need a pre-configured ConnectionFactory.
// Get it by looking it up using JNDI.
Context messaging = new InitialContext();
QueueConnectionFactory connectionFactory =
                                        (QueueConnectionFactory) messaging.lookup(" &");

// Gets Destination objects using JNDI.
Queue queue1 = (Queue) messaging.lookup(" &");
Queue queue2 = (Queue) messaging.lookup(" &");

// Creates QueueConnection, then use it to create a session.
QueueConnection connection = connectionFactory.createQueueConnection();
QueueSession session = connection.createQueueSession(
                                        false,  // not transacted
                                        Session.AUTO_ACKNOWLEDGE);      // acknowledge on receipt

// Getting a QueueSender and a QueueReceiver.
QueueSender sender = session.createSender(queue1);
QueueReceiver receiver = session.creatReceiver(queue2);

// Getting a QueueReceiver with a selector message.
Queue queue3 = (Queue) messaging.lookup(" &");
String selector = new String("(name = 'Bull') or (name = 'IBM'))");
QueueReceiver receiver2 = session.creatReceiver(queue3, selector);
```

Initialisation d'une session "*Publish/Subscribe*"

```
// We need a pre-configured ConnectionFactory.
// Get it by looking it up using JNDI.
Context messaging = new InitialContext();
TopicConnectionFactory connectionFactory =
                                        (TopicConnectionFactory) messaging.lookup(" &");
// Gets Destination object using JNDI.
Topic Topic = (Topic) messaging.lookup(" &");

// Creates TopicConnection, then use it to create a session.
TopicConnection connection = connectionFactory.createTopicConnection();
TopicSession session =
                                        connection.createTopicSession(false, Session.CLIENT_ACKNOWLED

// Getting a TopicPublisher  &
TopicPublisher publisher = session.createPublisher(Topic);
//  & then a TopicSubscriber.
TopicSubscriber subscriber = session.createSubscriber(Topic);
Subscriber.setMessageListener(listener);
```

Construction de messages

Les messages sont créés au moyen de primitives de l'interface `Session` en fonction du type de message désiré : `createBytesMessage`, `createTextMessage`, `createMapMessage`, `createStreamMessage`, et `createObjectMessage`.

Utilisation d'un `BytesMessage`

```
byte[] data
BytesMessage message = session.createByteMessage();
message.writeBytes(data);
Utilisation d'un TextMessage
StringBuffer data
TextMessage message = session.createTextMessage();
message.setText(data);
```

Utilisation d'un `MapMessage`

Chaque message est constitué d'un ensemble de paires noms/valeur. Le client reçoit la totalité du message, il peut n'extraire que la partie utile du message et ignorer le reste ; les paires sont positionnées au moyen de méthodes sur l'objet `Message`, l'ordre des mises à jour est quelconque.

```
MapMessage message = session.createMapMessage();
message.setString("Name", " &");
message.setDouble("Value", doubleValue);
message.setLong("Time", longValue);
```

Utilisation d'un `createStreamMessage`

De la même manière, le serveur peut envoyer un message contenant les différents champs en séquence, chacun écrit au moyen de sa propre primitive. Si le client n'est intéressé que par une partie du message il doit tout de même le lire entièrement. Les champs doivent être écrits dans l'ordre de lecture.

```
StreamMessage message = session.createStreamMessage();

message.writeString(" &");
message.writeDouble(doubleValue);
message.writeLong(longValue);
```

Utilisation d'un `ObjectMessage`

L'information peut être envoyée sous la forme d'un objet sérialisable contenant l'ensemble des informations utiles. Le client utilise alors l'interface de cet objet pour obtenir l'information utile.

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

Envoi et reception de messages

Point-to-Point

```
// Sending of all of these message types is done in the same way:
sender.send(message);
// Receiving of all of these message types is done in the same way. Here is how to
// receive the next message in the queue.
// Note that this call will block indefinitely until a message arrives on the queue.
StreamMessage message;
message = (StreamMessage)receiver.receive();
```

Publish/Subscribe

```
// Sending (publishing) of all of these message types is done in the same way:
publisher.publish(message);
// Receiving of all of these message types is done in the same way. When the
// client subscribed to the Topic, it registered a message listener. This listener
// will be asynchronously notified whenever a message has been published to
// the Topic. This is done via the onMessage() method in that listener class.
// It is up to the client to process the message there.
void
onMessage(Message message) throws JMSException {
        // unpack and handle the messages we receive.
        ...
}
```

Lecture d'un message

BytesMessage

```
byte[] data;
int length;
length = message.readBytes(data);
TextMessage
StringBuffer data;
data = message.getText();
MapMessage              // Note : l'ordre de lecture des champs est quelconque.
String name = message.getString("Name");
double value = message.getDouble("Value");
long time = message.getLong("Time");
StreamMessage   // Note : l'ordre de lecture des champs
                                // doit etre identique a l'ordre d'ecriture.
String name = message.readString();
double value = message.readDouble();
long time = message.readLong();
ObjectMessage
obj = message.getObject();
```

E.3 Architecture de Joram

Cette section approfondit quelques aspects fondamentaux de l'architecture de la plate-forme JORAM qui lui confèrent des propriétés uniques en matière de flexibilité et de scalabilité.

E.3.1 Principes directeurs et choix de conception

La caractéristique essentielle de la plate-forme JORAM est son architecture distribuée et configurable. L'architecture de base de JORAM est de type *snowflake*, c'est-à-dire qu'une plate-forme JORAM est constituée d'un ensemble de serveurs JORAM distribués, interconnectés par un bus à message. Chaque serveur gère un nombre variable de clients JMS. La répartition des serveurs et la distribution des clients sur les serveurs sont de la responsabilité de l'administrateur de la plate-forme JORAM en fonction des besoins de l'application. Ce choix constitue donc un premier niveau de configuration. Un deuxième niveau réside dans la capacité de localiser les destinations (`Queues` et `Topics`) en fonction des besoins (nous verrons plus loin que ce choix d'implantation a des conséquences importantes sur les propriétés de l'application : performance, évolutivité, etc.). Un dernier niveau de configuration concerne les paramètres de qualité de service associés au bus à messages(protocole de communication, sécurité, persistance, etc.). Le choix de mécanismes appropriés est ici le résultat d'un compromis entre le niveau de qualité de service souhaité et le coût de la solution correspondante.

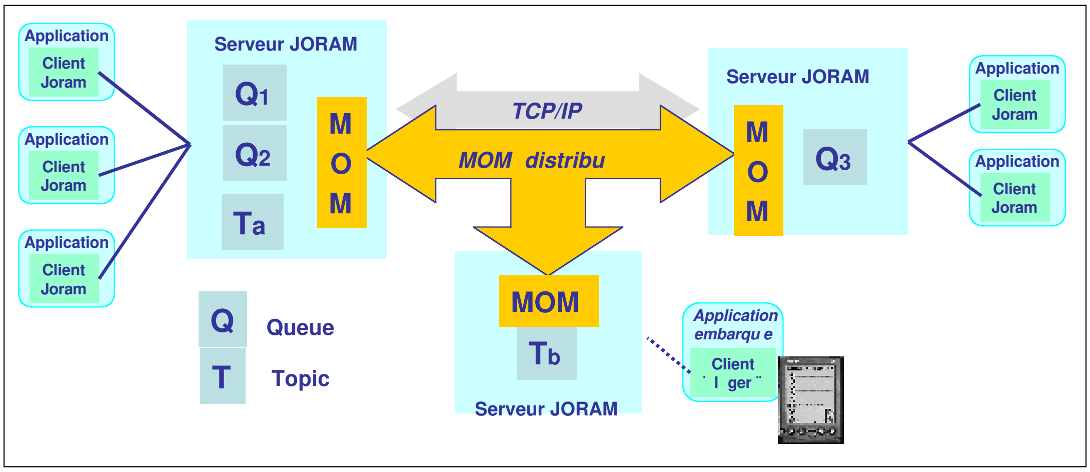

Figure E.7 – Architecture de la plate-forme JORAM

L'architecture générale d'une plate-forme JORAM est illustrée dans la figure 7.

E.3.2 Architecture logique

Le principe de fonctionnement d'une communication entre deux clients JMS est illustré dans la figure 8 (pour le cas particulier d'une communication point à point). Cette description permet de mettre en évidence les objets de communication et les flots de contrôle et de données. Les objets `Connection`, `Session` et `Sender` (respectivement `Receiver`) sont des objets temporaires JMS créés par le client JORAM lors de l'établissement d'une connexion logique entre le client et le serveur JORAM. Sur le serveur, chaque client est représenté par un objet `Proxy`. Cet objet persistant est créé par le serveur lors de la création d'un utilisateur. Il remplit deux fonctions essentielles :

- La gestion des communications entre le client et le serveur.
- L'acheminement des messages vers / depuis la destination (ici une queue de messages).

Modèle de communication point à point

A ce stade de la présentation, la distribution n'est pas représentée. Il s'agit d'un schéma purement fonctionnel (i.e. dans les faits, les objets `Proxy` et `Queue` peuvent être implantés sur un ensemble de serveurs coopérants). Une communication entre un client producteur et un client consommateur met en œuvre les échanges suivants (représentés par les flêches rouges) :

1. L'interprétation d'une primitive `Send` d'un client JMS producteur se traduit par l'envoi d'un message sur la connexion entre le client JORAM et l'objet `Proxy` (noté ici `Proxy-P`) associé au client dans le serveur JORAM. Cette interaction est représentée par la flèche pleine 1.
2. L'objet `Proxy-P` encapsule le message JMS dans un message destiné à être véhiculé par le bus (noté ici Message MOM). Ce message est sauvegardé dans un support persistant local. Un accusé de réception est envoyé au client JORAM qui le remonte ensuite au client JMS (flèche pointillée 2). Du point de vue du client JMS, l'opération `Send` est terminée.
3. Le message construit par l'objet `Proxy` est véhiculé par le MOM vers le site de résidence de la queue de messages. Cette interaction est représentée par la flèche pleine 3. Sur le site où se trouve la queue de messages, le message est enregistré sur un support persistant local.
4. L'interprétation d'une primitive `Receive` par le client consommateur génère un message de contrôle vers l'objet `Proxy` associé (noté ici `Proxy-C`) qui le fait suivre au site où se trouve la file de message (objet `Queue`). Le message applicatif correspondant est extrait de la queue de messages et est envoyé vers l'objet `Proxy-C`. Celui-ci extrait le message JMS de son enveloppe véhiculée par le MOM et l'envoie au client JMS. Cette suite d'échanges est représentée par les flèches pleines 4. La requête `Receive` du client consommateur est terminée.
5. Un accusé de réception est envoyé vers le site de la localisation de l'objet `Queue` afin de retirer le message de la queue et de libérer les ressources correspondantes (flèches pointillées). L'accusé de réception peut être généré par le client JMS ou par le système.

Dans l'étape N°2 le message MOM est sauvegardé avant d'être envoyé vers le site de l'objet `Queue` . En cas de problème pendant l'étape N°3, la sauvegarde réalisée par l'objet `Proxy` permet de ré-exécuter cette étape jusqu'à son aboutissement. Cette fonction, communément appelée *Store and Forward*, permet d'assurer la garantie de délivrance des messages au niveau du serveur JMS. Notons que nous ne décrivons pas ici les échanges au niveau MOM.

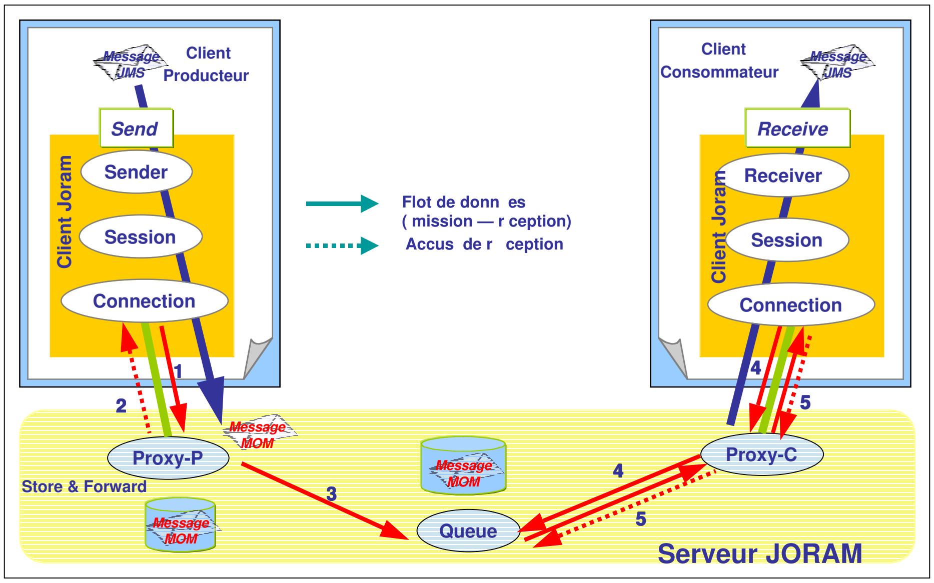

Figure E.8 – Communication Point à Point

Modèle de communication *Publish/Subscribe*

La figure 9 décrit les objets et flots de contrôle pour un mode de communication de type *Publish/Subscribe*.

1. Les étapes 1 et 2 sont similaires à celles du modèle de communication point à point.
2. Dans l'étape 3, le message construit par le `Proxy` du producteur est acheminé directement vers l'ensemble des objets `Proxy` des consommateurs où ils sont enregistrés. Notons ici une différence essentielle avec le schéma précédent dans la mesure ou l'objet `Topic` est utilisé, non pas comme une destination finale, mais plutôt comme un aiguillage vers un ensemble de destinations finales représentées par les objets `Proxy` des consommateurs.
3. L'opération de consommation du message se traduit par un échange entre le client et l'objet `Proxy-C` (flèches pleines 4). L'accusé de réception est ensuite envoyé vers l'objet `Proxy-C` (flèche pointillée 5). Par rapport au schéma précédent on notera que le protocole de consommation se limite à un dialogue entre le client et l'objet `Proxy`. Il n'y a pas de dialogue entre les objets `Proxy` et le `Topic` .

Dans la suite, on étudie comment ce schéma d'architecture logique est mis en œuvre dans le cadre d'une configuration centralisée (serveur JORAM unique) dans un premier temps, puis dans une configuration distribuée (serveurs coopérants).

E.3.3 Architecture centralisée

Cette option correspond à une configuration simple dans laquelle tous les objets `Destination` sont centralisés sur un seul serveur auquel sont connectés des clients par l'intermédiaire de l'objet `Proxy` correspondant. Cette configuration est illustrée dans la figure 10.

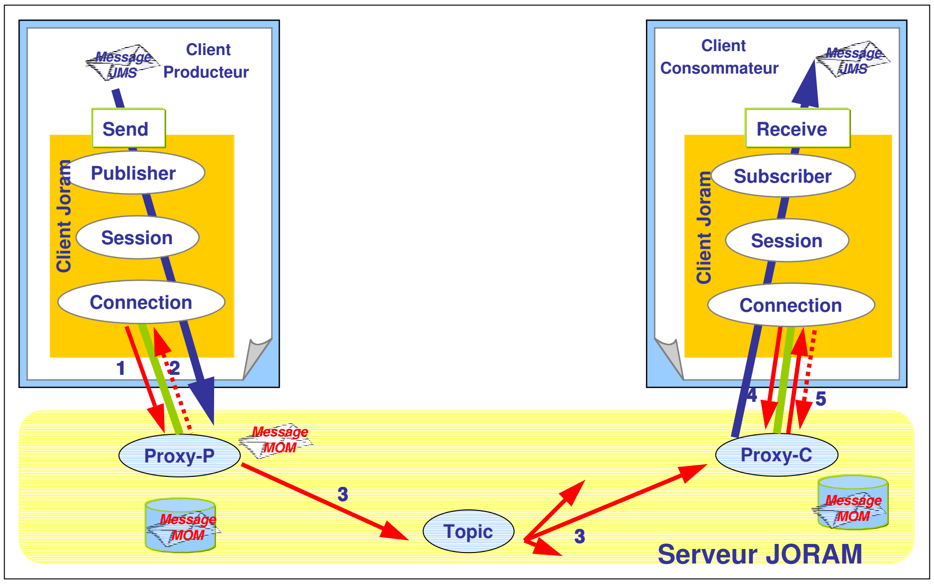

Figure E.9 – Modèle de communication *Publish/Subscribe*

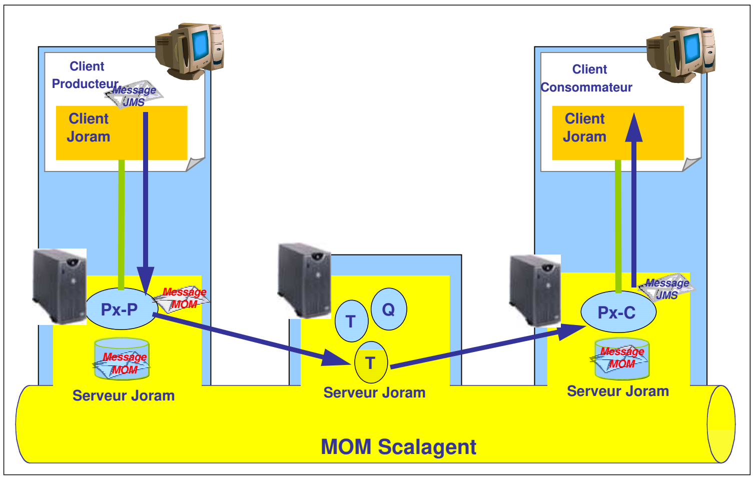

Figure E.10 – Architecture centralisée

Dans cette figure, les flots liés à l'opération de production sont représentés par des flèches pleines. Les flots de l'opération de consommation sont représentés par des flèches pointillées. On ne représente pas les flots liés aux accusés de réception. Dans le cas d'une communication de type *Publish/Subscribe* le dialogue est réalisé directement entre les deux objets `Proxy`.

Dans cette configuration la simplicité des échanges est une conséquence directe de la co-localisation de l'objet `Queue` et des objets `Proxy`. Un autre élément de simplicité est lié aux opérations d'administration (i.e. création des utilisateurs, des destinations, etc.) qui sont regroupées sur un site unique. L'inconvénient majeur de cette solution est son manque de disponibilité et une capacité d'extension réduite. Une défaillance du serveur signifie un arrêt du système global. Par ailleurs le nombre de requêtes et d'objets gérés par le serveur est limité par sa capacité de calcul et de stockage.

E.3.4 Architecture distribuée

Dans une architecture distribuée, plusieurs serveurs JORAM coopèrent pour assurer la communication des messages entre des clients connectés à ces divers serveurs. La Figure 11décrit la structure des échanges pour une architecture répartie sur trois sites géographiques composés chacun d'un serveur et d'un client. Le schéma représenté utilise l'exemple d'une communication point à point dans lequel chaque serveur est responsable de la queue de messages destinés aux clients JMS connectés à ce serveur[5].

Le message JMS, émis par le client JMS *Client* à destination de la queue de messages *Q3* est envoyé vers le proxy Px1 sur le serveur *S1*. Le message MOM qui encapsule le message JMS est généré. La fonction *Store and Forward* sauve le message localement avant de l'envoyer vers le serveur *S3*, où se trouve localisée la queue *Q3*. Le message stocké dans la queue *Q3* peut être consommé par la suite par un client JMS connecté au serveur *S3*. De façon identique un message produit par le client JMS *Client-2* à destination de la queue de messages *Q1* est transmis via l'objet *Px2* du serveur *S2*, puis est ensuite consommé par le client JMS *Client* à partir du serveur local *S1*.

La communication entre les serveurs est mise en œuvre par le bus à messages sous-jacent qui garantit l'acheminement des messages quelles que soit les pannes de sites et de réseau.

E.3.5 Joram : une plate-forme JMS configurable

La construction d'une plate-forme JMS adaptée à un contexte applicatif donné est un exercice difficile dans la mesure où l'architecture de la plate-forme est le résultat de compromis délicats entre de nombreux critères souvent antinomiques, tels que performance, disponibilité, scalabilité, flexibilité et capacité d'évolution, sécurité, coûts de développement et d'exploitation, etc. Le rôle de l'architecte dans la définition d'une plate-forme est essentiel. À partir d'un cahier des charges il doit prendre en compte l'ensemble des critères d'évaluation et leur fixer un poids dans la perspective de définir l'architecture « la mieux adaptée ». Ce travail n'est possible que si le service de messagerie fournit des capacités de configuration suffisantes pour traduire les options d'architecture retenues. C'est le cas de

[5]Cette configuration particulière est prise à titre d'exemple. Dans la réalité la configuration des `Queues` et `Topics` sur les serveurs est laissée à l'initiative de l'administrateur.

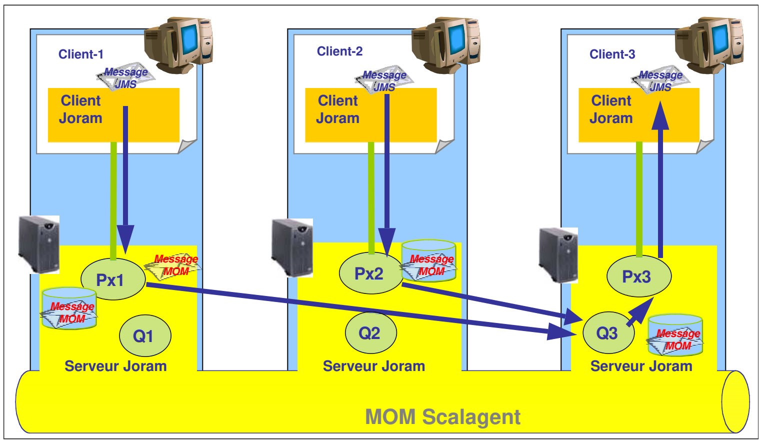

Figure E.11 – Architecture distribuée

la plate-forme Joram qui permet, à plusieurs niveaux, de définir les choix de conception appropriés :

- Organisation des serveurs et clients JMS, et placement des objets JMS. Comme cela est illustré dans la figure 7, La structure de la plate-forme Joram est de type *snowflake*, ce qui donne à l'architecte du système une grande liberté pour décider de l'emplacement des serveurs et de la répartition des clients sur les serveurs pour répondre aux besoins de l'application (par exemple pour servir un ensemble de clients géographiquement proches dans une approche de type serveur de proximité, ou bien pour respecter certaines contraintes de sécurité). Multiplier le nombre de serveurs pour une meilleure couverture géographique des clients distribués a un coût d'exploitation (en terme de machines) qui doit être mis en balance avec la performance et la fiabilité des communications pour des clients très éloignés d'un serveur. Le dimensionnement des serveurs (calcul, mémoire, stockage) est également un point clé pour la performance et la disponibilité de l'ensemble du système.
- Placement des objets de communications (`Queues` et `Topics`). Sauf dans des cas très particuliers, on peut noter qu'il est souhaitable de rapprocher les objets Destination des clients consommateurs. Cette stratégie a un impact positif sur les performances et la disponibilité des clients.
- Evolution du système et passage à l'échelle. Cette propriété fait référence à la capacité de faire évoluer le système pour répondre aux évolutions de l'application. Joram fournit ainsi des fonctions d'administration qui permettent d'ajouter et/ou de retirer un serveur depuis un point central d'administration.
- Protocoles de communication. Plusieurs protocoles de transport sont disponibles pour la connexion client - serveur et pour les connexions entre serveurs : TCP/IP, HTTP, SOAP, SSL, etc.
- Paramètres de qualité de service (persistance, sécurité). La fiabilité et la mise en sécurité des communications a un coût. C'est pourquoi Joram donne la possibilité de retenir ou non ces options selon les besoins.
- Niveau de disponibilité grâce à la clustérisation et à la réplication (voir plus loin en section 4).

Peu de systèmes aujourd'hui permettent d'agir simultanément sur ces divers paramètres et de construire ainsi la plate-forme de messagerie adaptée à un environnement particulier. Le niveau de flexibilité de JORAM est de ce point de vue un atout incontestable par rapport aux produits concurrents.

E.4 Joram : fonctions avancées

Cette section décrit quelques fonctions avancées de JORAM (au sens où elles ne sont pas définies dans la spécification JMS 1.1).

E.4.1 Répartition de charge

L'architecture distribuée de JORAM est exploitée pour mettre en oeuvre des mécanismes de répartition de charge avec le double objectif suivant : accroître la disponibilité grâce à la réplication des objets de communication ; optimiser le flux des messages entre les serveurs. La répartition de charge s'applique de façon différente aux `Topics` et aux files de messages (`Queues`).

`Topic` répliqué

Un « `Topic` clustérisé » est répliqué sur plusieurs serveurs. Notons que cette forme de réplication s'applique aussi bien à des serveurs fortement couplés (grappe de machines) qu'à des serveurs géographiquement distribués. Le principe de fonctionnement du `Topic` clustérisé est illustré dans la figure 12.

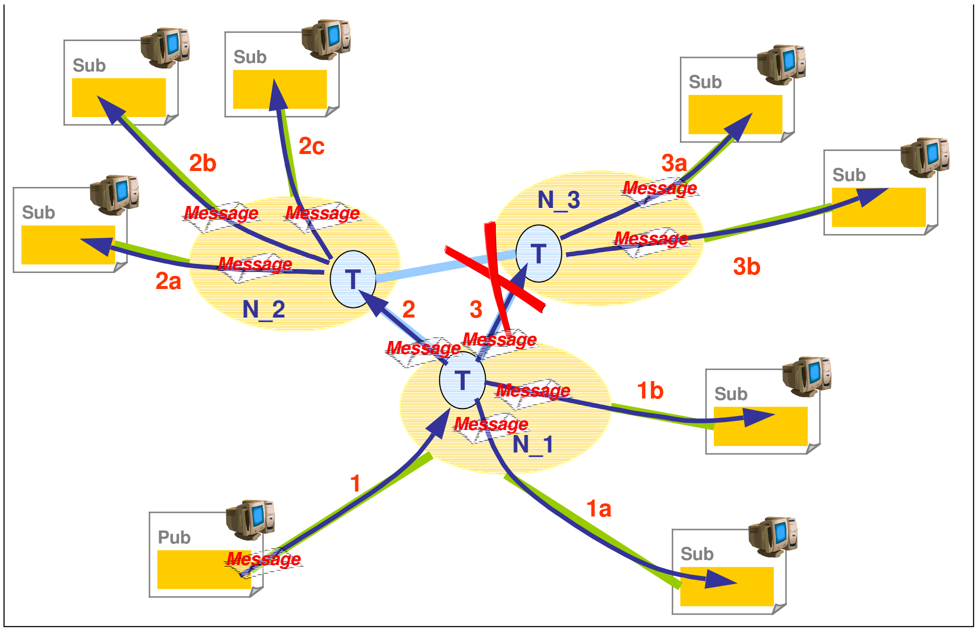

Figure E.12 – `Topic` "clustérisé"

Dans cet exemple un `Topic` (noté T) est répliqué sur trois serveurs N_1, N_2 et N_3. Des clients JMS connectés à ces serveurs sont supposes s'être abonnés au `Topic` T. Chaque serveur est responsable de la gestion des abonnements réalisés par ses propres clients. Une application cliente sur le nœud N_1 publie un message correspondant au `Topic` T (flot noté 1 dans la figure 12). Le nœud N_1 diffuse le message à ses abonnés locaux (flots 1a et 1b) et fait suivre le message aux nœuds N_2 et N_3 (flots 2 et 3). Par la suite, chacun d'entre eux diffuse le message reçu à ses clients locaux (flots 2i et 3j).

La mutualisation des messages entre les nœuds permet de réduire le trafic. Par ailleurs, une panne d'un serveur n'affecte que les clients connectés à ce serveur. Le reste de l'application continue à fonctionner. L'utilisation d'un `Topic` clustérisé est transparent du point de vue du programmeur d'application mais requiert beaucoup d'attention de la part de l'administrateur du système pour être efficace.

`Queue` répliquée

Le principe des « queues clustérisées » est un peu différent (voir figure 13). Plusieurs exemplaires du même objet `Queue` sont localisés sur des serveurs indépendants. Chacune de ces copies est accessible uniquement aux clients connectés au serveur correspondant. Si la charge sur une copie dépasse un certain seuil, les messages reçus ensuite sur cette copie sont redirigés vers un autre exemplaire de la même queue géré par un autre serveur. Le seuil est un paramètre configurable qui peut prendre en compte divers critères tels que : nombre de messages en attente, nombre de requêtes de lecture en attente, délai d'attente dépassé pour une requête en lecture, etc. Il est important de noter que la sémantique des queues de message n'est pas modifiée, à savoir qu'un message donné n'est consommé que par un seul client JMS).

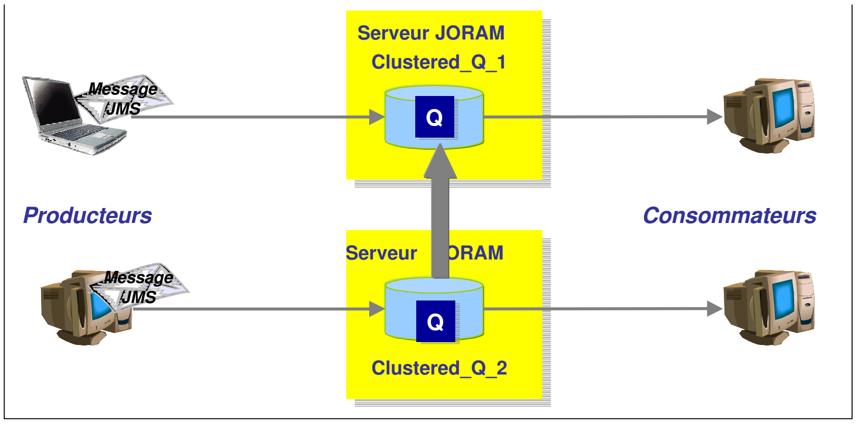

Figure E.13 – *Clustered Queue*

Comme dans le cas des `Topic` , le concept de queue clustérisée permet d'améliorer les performances et le niveau de disponibilité sans impact sur la programmation de l'application.

E.4.2 Fiabilité et haute disponibilité

Le terme fiabilité fait référence ici à la capacité de transporter un message de bout en bout entre un client producteur et un client consommateur malgré les incidents pouvant affecter de façon temporaire le réseau et les serveurs. La fiabilité dans JORAM est la résultante de plusieurs mécanismes complémentaires :

- Un accusé de réception entre un client JMS et son représentant dans le serveur (objet *proxy*) permet de fiabiliser la communication entre client et serveur (sémantique « au moins une fois »).
- La fonction Store and Forward réalisée par le proxy permet de fiabiliser les échanges entre le *proxy* et la destination (sémantique « exactement une fois »).
- Enfin le bus à messages assure des échanges fiabilisés entre serveurs (voir 5.2).

Pour répondre aux besoins de haute disponibilité une version spécifique du serveur JORAM (notée JORAM HA pour *High Availability*) a été conçue en suivant une approche de réplication active en mode Maître - Esclave. Le schéma de principe du serveur JORAM HA est représenté sur la figure 14.

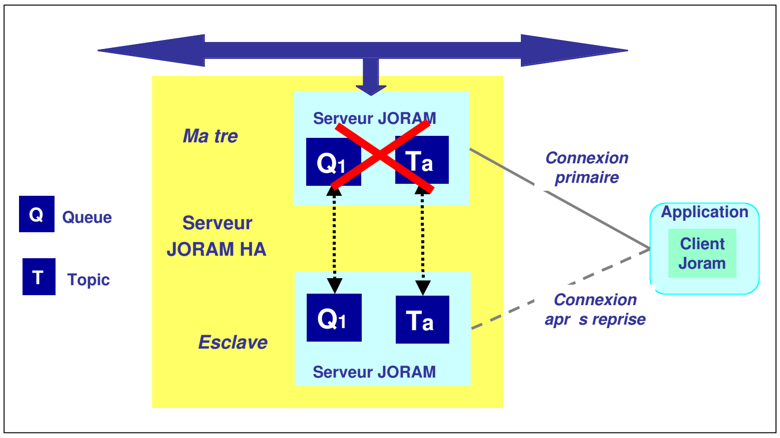

Figure E.14 – Serveur JORAM HA

`Queues` et `Topics` sont répliqués sur des serveurs JORAM s'exécutant sur une grappe de machine. Le serveur maître exécute les requêtes des clients et propage les opérations vers le serveur esclave qui réplique le traitement localement. En cas de panne du serveur maître, les clients établissent une nouvelle connexion vers le serveur esclave et continuent leur travail sans interruption. Ce fonctionnement permet un haut niveau de continuité de service au prix d'une redondance du serveur JORAM. La version actuelle du serveur JORAM HA utilise les mécanismes de JGroups.

E.4.3 Connectivité élargie

Cette section décrit plusieurs fonctions qui permettent d'enrichir la connectivité entre une plate-forme JORAM et le monde extérieur.

- Passerelle JMS pour l'interopérabilité avec d'autres plates-formes JMS,
- Utilisation du protocole SOAP
- Support de l'architecture de connectivité JCA 1.5

– Passerelles vers les protocoles SMTP et FTP

Passerelle JMS

La passerelle JMS permet à une application JMS gérée par JORAM de communiquer avec une destination gérée par une autre plate-forme JMS (appelée xMQ dans la figure 15) de façon transparente du point de vue du programmeur d'application.

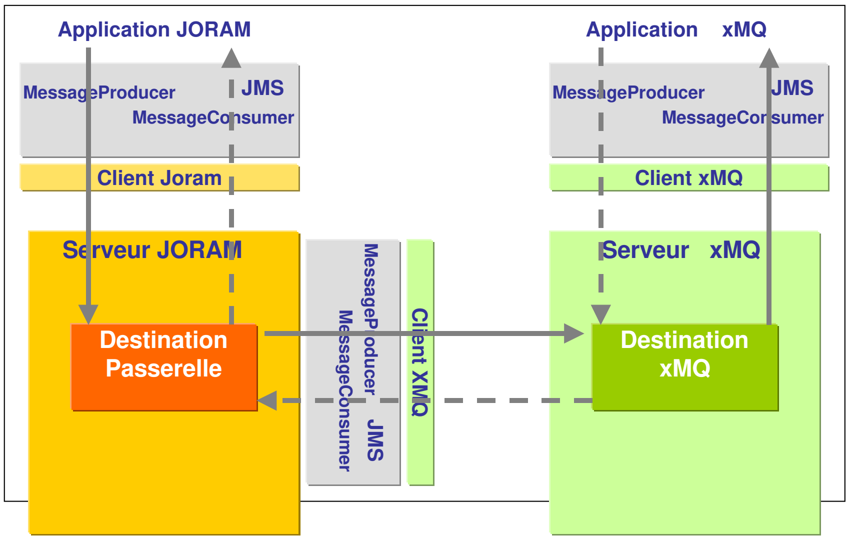

Figure E.15 – Passerelle JMS

Le lien entre une plate-forme JORAM et une plate-forme xMQ est réalisé par le biais d'un objet destination JORAM spécifique, appelé destination passerelle (qui peut être une `Queue` ou un `Topic`), qui est le représentant de la destination finale. L'objet « destination passerelle » joue deux rôles complémentaires :

– En tant que destination gérée par JORAM, il reçoit les messages produits par les clients producteurs et les requêtes des clients consommateurs gérés par la plate-forme JORAM.
– En tant que représentant d'un objet destination externe, il se comporte comme un client JMS géré par la plate-forme xMQ pour propager les messages et requêtes vers la destination finale.

Utilisation du protocole SOAP

L'utilisation du protocole HTTP-SOAP fournit un moyen normalisé pour accéder à des services distants en échangeant des messages XML sur des connexions HTTP. Dans certains cas il peut être utile d'utiliser ce protocole pour accéder depuis un client aux services d'un serveur JORAM, en particulier pour répondre aux besoins suivants :

– Contraintes de sécurité imposées par la gestion de pare-feux,
– Prise en compte de clients s'exécutant dans un environnement J2ME pour lequel l'API JMS complète ne peut pas être fournie (par exemple J2ME ne fournit pas de

fonction de sérialisation des objets).

L'usage du protocole SOAP dans JORAM est illustré dans la figure 16. Sur le serveur, un objet proxy particulier, construit comme un service SOAP, fournit un accès à des clients SOAP. Le service SOAP utilisé par JORAM est la version développée par la fondation Apache (http ://ws.apache.org/soap). Dans cette approche un serveur JORAM est encapsulé dans un conteneur Tomcat et agit comme une passerelle entre Tomcat et les autres services de JORAM. Du côté client, une bibliothèque spécifique met en œuvre une connexion basée sur le protocole HTTP et des messages au format XML/SOAP. Dans les environnements J2ME, une version particulière de cette bibliothèque, appelée KJORAM, a été adaptée pour prendre en compte les restrictions imposées par J2ME.

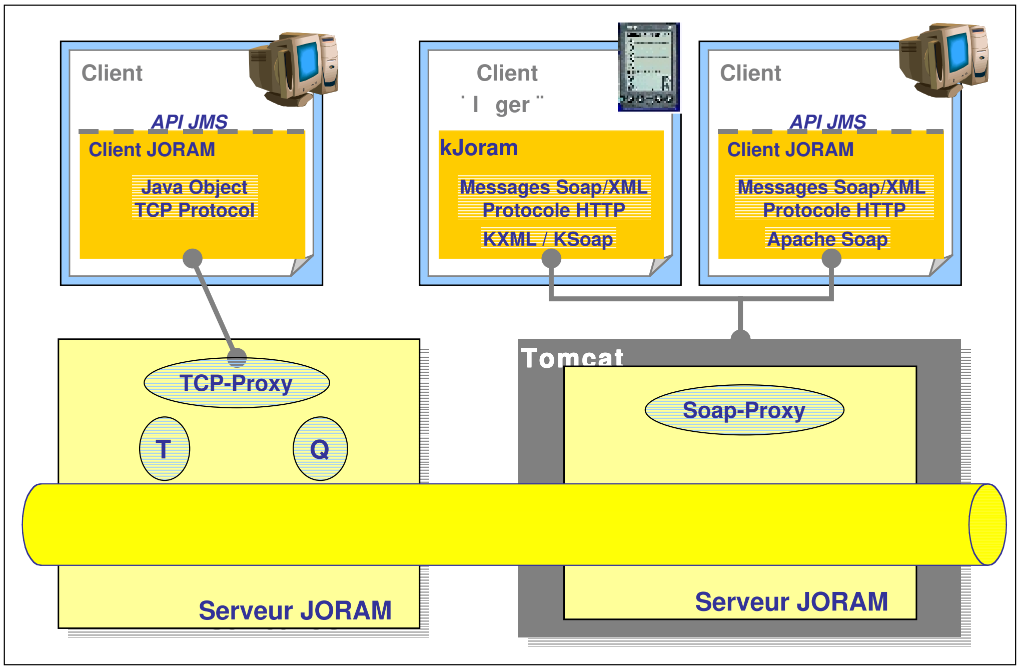

Figure E.16 – Utilisation de SOAP et KJORAM

La disponibilité de KJORAM étend les usages de la plate-forme JORAM à de nouveaux domaines d'application, marqués en particulier par l'utilisation d'équipements disposant de ressources limitées (par exemple les PDA, les téléphones et de façon plus générale tous les terminaux Java). Les applications J2ME s'exécutant sur ces équipements sont maintenant capables de coopérer avec des applications JMS s'exécutant sur des serveurs d'applications.

JCA 1.5

JORAM est conforme aux spécifications de l'architecture JCA 1.5 (*J2EE Connector Architecture*), qui décrit la manière d'intégrer des ressources externes dans un serveur d'application J2EE. Les fonctions suivantes sont disponibles :
– Gestion du cycle de vie de la ressource : création, démarrage, arrêt.
– Gestion des connexions avec les composants EJB

– Gestion transactionnelle conforme à l'interface XA.

L'utilisation de l'API JCA permet d'intégrer JORAM avec tout serveur d'application J2EE qui met en oeuvre cette spécification. C'est en particulier la voie d'intégration classique avec le serveur JOnAS (http ://jonas.objectweb.org).

Connecteurs Joram

Les connecteurs JORAM offrent des passerelles pour réaliser l'interopérabilité avec des applications externes en utilisant des protocols de transport normalisés. Deux connecteurs sont disponibles aujourd'hui :

- Passerelle *mail* : cette fonction permet d'émettre et recevoir des messages JMS en utilisant le protocole SMTP. La fonction est mise en oeuvre par le biais d'objets `Queues` et `Topics` spécialisés pour réaliser cette interopérabilité. Ces objets sont configurés et installés comme les destinations normales.
- Passerelle FTP : cette fonction est identique à la précédente pour le protocole FTP. C'est un dispositif utile lorsque les messages JMS à transporter sont de très grande taille.

E.4.4 Sécurité

Pour sécuriser les échanges (entre serveurs et entre un serveur et ses clients), JORAM utilise, à la demande, des connexions SSL afin d'authentifier les acteurs et de chiffrer les messages. La gestion des pare-feux pose des problèmes particuliers. La stratégie recommandée consiste à configurer les pare-feux pour autoriser l'usage des ports requis par JORAM. Cette solution s'applique autant pour les communications client - serveur que pour les communications entre serveurs. Une solution alternative consiste à utiliser, dans la plate-forme JORAM, des protocoles couramment acceptés par les pare-feux (HTTP et SOAP). La section 4.3.2 a montré comment SOAP peut être utilisé pour sécuriser la liaison entre clients et serveurs. Les serveurs JORAM disposent également d'un module de communication fondé sur HTTP pour permettre les communications entre serveurs traversant un pare-feu.

E.5 Joram : bases technologiques

Les propriétés de JORAM sont une conséquence directe de la technologie utilisée pour la réalisation de la plate-forme. Cette technologie et l'usage qui en est fait dans JORAM sont brièvement présentés dans cette section.

E.5.1 Les agents distribués

L'expérience de nombreuses années d'étude des architectures réparties nous a conduit à adopter une approche de type « intelligence distribuée » pour la conception des applications réparties. Rapprocher les traitements des sources de données permet, d'une part de répartir la charge de travail sur les ressources de calcul disponibles sur le réseau, et d'autre part de limiter la bande passante en ne véhiculant que l'information pertinente.

La mise en œuvre de ce principe fondateur s'appuie sur un modèle de programmation, un environnement d'exécution, et des outils de développement présentés dans la suite.

– Modèle de programmation à base d'agents communicants. Les agents sont des objets Java distribués communiquant par messages. Le contenu d'un agent est défini par une classe Java héritant d'une classe pré-définie « agent » qui définit le comportement générique d'un objet agent. Les agents se conforment à un modèle de programmation de type « événement-réaction ». Un événement correspond à la notification d'un message typé qui va se traduire par l'exécution d'une méthode de l'objet. Cette exécution peut, à son tour, provoquer la production d'événements auxquels un ou plusieurs agents vont réagir. Par défaut l'exécution d'une réaction est atomique (i.e. propriété du "tout ou rien") et l'état d'un agent est persistant.
– Environnement d'exécution : serveurs d'agents. Les agents s'exécutent au sein d'une structure d'accueil nommée « serveur d'agents », dont l'organisation est représentée sur la figure 17.

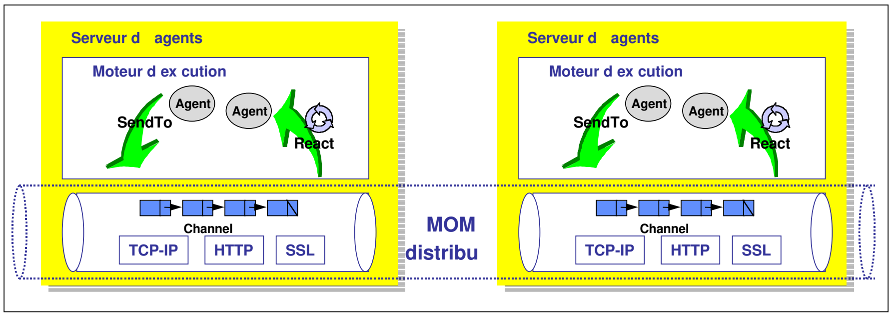

Figure E.17 – Structure d'un serveur d'agents

Le cœur du serveur est un moteur d'exécution qui contrôle le flot d'exécution des agents. Ce flot est unique et prend la forme d'une boucle qui consiste, pour chaque notification, à exécuter la méthode associée à la réaction de l'agent destinataire. Le flot exécute aussi le code des modules de persistance et d'atomicité. Le serveur comporte également des flots d'exécution associés à la gestion des communications au sein d'un sous-système appelé le bus local. Les bus locaux des différents serveurs coopèrent pour fournir une mise en œuvre distribuée d'un bus à message distribué. Ce bus à messages global permet la communication entre les différents agents, qu'ils soient hébergés par le même serveur ou non. Un bus local est constitué de deux types de composants : un « canal » et un ensemble de « composants réseau ».

– Le canal est chargé de distribuer les messages en provenance du moteur d'exécution vers les composants réseau et vice-versa.
– Les composants réseau sont responsables de l'émission de messages provenant du canal à destination des serveurs distants. Il existe différents composants réseau correspondant à différents protocoles de communication (par exemple TCP/IP, HTTP, SOAP, etc.). Par ailleurs, il est possible d'ajouter à un composant réseau des modules logiciels pour la mise en œuvre de propriétés complémentaires (par exemple sécurité, ordonnancement causal, etc.).

La structure d'accueil est configurable et fournit différentes politiques de fonctionnement des agents hébergés, en particulier l'atomicité des réactions (i.e. stratégie du « tout ou rien »), et la persistance de l'état des agents (i.e. un changement d'état correspond à la complétion d'une réaction) Un ensemble d'opérations d'administrations élémentaires permettent de créer et de configurer des serveurs d'agents dans un environnement Internet.

E.5.2 Joram : une plate-forme JMS à base d'agents

La technologie à base d'agents distribués décrite ci-dessus a été largement utilisée pour construire des systèmes répartis dans des domaines d'application très variés. La plate-forme Joram est un système distribué particulier mis en œuvre à l'aide de cette technologie. Un serveur Joram, c'est-à-dire la partie de la plate-forme Joram qui implante les objets JMS est un serveur d'agents structuré de la façon suivante :

- `Queues` et `Topics` sont représentés par des agents persistants.
- Sur chaque serveur un agent `ConnectionManager` gère les connexions avec les clients JMS gérés par ce serveur.
- Un agent Proxy persistant est créé pour chaque utilisateur reconnu par le système. Par la suite, cet agent a la charge de gérer la communication pour le compte des clients JMS (producteur ou consommateur) associés à cet utilisateur. Rappelons qu'une de ses fonctions essentielles est de mettre en oeuvre la fonction *Store and Forward*.

La persistance des abonnements et des messages est mise en œuvre directement par la persistance de l'état des agents correspondants. Notons que cette propriété est « débrayable » si elle n'est pas requise par l'application, ce qui se traduit par un gain de performance. La flexibilité de la plate-forme Joram bénéficie directement de la capacité de configuration des serveurs d'agents, en particulier pour ce qui concerne les éléments suivants :

- Le choix des protocoles de communication (TCP/IP, HTTP, SOAP, etc.) et des paramètres de QoS (persistance et garantie de délivrance, sécurité, etc.).
- Un choix d'implantation des serveurs Joram pour définir une configuration distribuée de type « snowflake » qui réponde aux besoins de l'application cible.
- Un choix d'implantation des objets (agents) `Destination` et des objets (agents) `Proxy`.

Les services d'administration permettent, par la suite, de faire évoluer ces paramètres de configuration pour adapter la structure d'un système Joram à de nouveaux besoins (implantation d'un nouveau serveur, migration des `Queues` et des `Topics` , changement de paramètres de sécurité, etc.).

E.6 Conclusion

Joram est un système de messagerie caractérisé par deux propriétés majeures :

- Il est conforme à la spécification JMS1.1. Cette conformité a été démontrée dans le cadre de la campagne de certification J2EE 1.4 réalisée en liaison avec le serveur d'application JOnAS[6]. Il est donc possible d'utiliser Joram comme environnement d'exécution de toute application distribuée qui suit l'interface de programmation définie par JMS.

[6]La spécification JMS est un élément de la spécification J2EE.

– L'architecture de JORAM est distribuée et hautement configurable. Cette propriété permet de concevoir une plate-forme d'exécution adaptée aux besoins d'une application donnée.

La figure 18 synthétise les divers usages de JORAM pour la mise en œuvre d'applications distribuées.

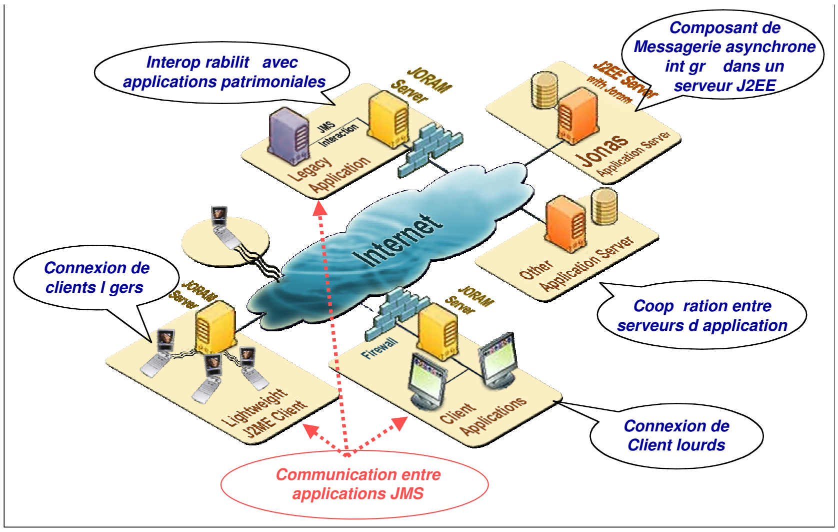

Figure E.18 – Les usages de JORAM

Ces usages sont les suivants.

– Système de communication asynchrone fiable entre applications JMS s'exécutant dans des environnements Java variés (J2EE, J2SE, J2ME), y compris la possibilité d'établir une passerelle vers des applications s'exécutant sur des plates-formes JMS externes.
– Composant de messagerie asynchrone intégré dans un serveur d'application J2EE. L'intégration dans le serveur JOnAS (http ://jonas.objectweb.org) est la plus accomplie (administration globale unique). JORAM est également utilisé en liaison avec d'autres serveurs J2EE, en particulier JBoss.
– La combinaison des deux usages précédents permet d'élargir le champ d'action des serveurs d'application à des clients s'exécutant sur des dispositifs à faibles ressources (téléphones mobiles, équipements industriels, etc.).
– Interopérabilité avec des applications patrimoniales (legacy applications). JMS est aujourd'hui le canal de communication privilégié pour l'intégration d'application dans les plates-formes EAI (*Enterprise Application Integration*) et ESB (*Enterprise Service Bus*). En conséquence JORAM peut être utilisé comme base de mise en œuvre d'une telle plate-forme d'intégration.
– Coopération entre serveurs d'application J2EE. L'utilisation de JORAM permet ainsi de réaliser des versions distribuées du serveur d'application JOnAS.

La plate-forme JORAM est aujourd'hui en exploitation opérationnelle dans le monde entier pour des contextes applicatifs très variés : santé, banque, transport, logistique, télécommunications, etc. Parmi les usages connus[7] on peut citer l'EDI, l'intégration de données, l'administration de systèmes et d'applications répartis.

Les travaux en cours sur Joram visent deux objectifs complémentaires :

- Etendre son champ d'application, en particulier dans le domaine des systèmes embarqués. Cet objectif requiert de pouvoir réaliser une version « légère » du serveur Joram pour répondre aux contraintes de ressources de nombreuses classes d'équipements tels que carte à puce, lecteur RFID, contrôleur industriel, etc (en particulier pour ce qui concerne la mémoire RAM et la mémoire persistante de type mémoire flash) . Une approche, parmi d'autres, consiste à déporter une fonction store and forward dans la librairie client. Cette nouvelle structure permettrait également de mettre en œuvre, sous certaines conditions, des liaisons pair à pair (*peer to peer*) entre clients Joram sans avoir recours à un serveur tiers.
- Améliorer les performances et les fonctions d'administration de la plate-forme Joram. Ces deux aspects sont présentés globalement dans la mesure où ils présentent des dépendances fortes. La performance est un paramètre difficile à maîtriser car il dépend de critères multiples, tels que volume et taille des messages, nombre de clients, nombre de connexions, etc. L'optimisation d'une plate-forme Joram relève à la fois de l'évolution de certains mécanismes internes (gestion de la concurrence, de la mémoire et de la persistance pour ne citer que ceux là), et du bon dimensionnement de la plate-forme pour un usage déterminé (par exemple les ressources des serveurs, le placement des objets de communication, etc.). Parmi les pistes explorées, l'usage d'un système de gestion de base de données pour gérer la persistance des messages est envisagé. Une piste à plus long terme concerne l'usage de mécanismes auto-adaptatifs pour permettre à une plate-forme Joram de recueillir des informations sur son comportement afin de s'adapter « spontanément » à des défaillances (auto-réparation), à des baisses de performance (auto-optimisation), ou plus simplement à des changements de configuration (connexion / déconnexion d'un serveur).

Un axe de travail complémentaire concerne l'utilisation de Joram comme moteur d'un bus de services d'entreprise (ou ESB pour *Enterprise Service Bus*). L'objectif est d'enrichir Joram avec des fonctions de transformation de données (fondées sur une représentation XML) et de routage par le contenu pour répondre aux besoins des utilisateurs en matière d'intégration d'applications.

Pour conclure, notons que Joram est une brique d'une plate-forme technologique plus large qui vise à fournir les fondations techniques pour la construction et le déploiement de solutions d'intégration dans le monde Internet/Java (voir figure 19).

Comme cela a été exposé dans la section 5, Joram a été réalisé en s'appuyant sur la technologie à base d'agents développée par ScalAgent. En d'autres termes on peut considérer que Joram constitue une personnalité d'un système réparti à agents qui met en œuvre l'API JMS. Notons que l'ensemble de c'est l'ensemble du produit « agents + Joram » qui est est disponible en logiciel libre, comme indiqué dans la figure 19.

Sur la même base technologique, ScalAgent a développé des briques d'intégration complémentaires qui s'appuient sur deux technologies émergentes : les composants et les architectures logicielles. L'intérêt pour les composants est une réalité aujourd'hui comme

[7]Comme la plupart des produits diffusés en logiciel libre, les retours connus sur les usages de Joram sont peu nombreux et, en règle générale, peu détaillés.

le montre l'usage croissant des architectures à base d'EJB ou .NET. Par ailleurs les composants constituent des unités de programmation à gros grain qui facilitent la réutilisation et la construction d'applications par assemblage (i.e. sans recours à un langage de programmation). Cet aspect, qui vise à simplifier et raccourcir le cycle de développement des applications, n'est pas encore intégré dans les modèles tels que EJB et .NET. Cette capacité de construction d'applications par assemblage de composants est exploitée par les approches fondées sur les architectures logicielles. La définition des composants, de leurs interfaces et des leurs dépendances fonctionnelles est décrite dans un langage déclaratif, appelé ADL (pour *Architecture Description Language*) qui prend la forme d'un IDL étendu. La compilation d'une description ADL fournit une représentation logique de l'application distribuée. Ce référentiel est ensuite exploité dans toutes les étapes du cycle de vie de l'application pour automatiser et contrôler les opérations d'administration telles que le déploiement, la surveillance et la reconfiguration de l'application.

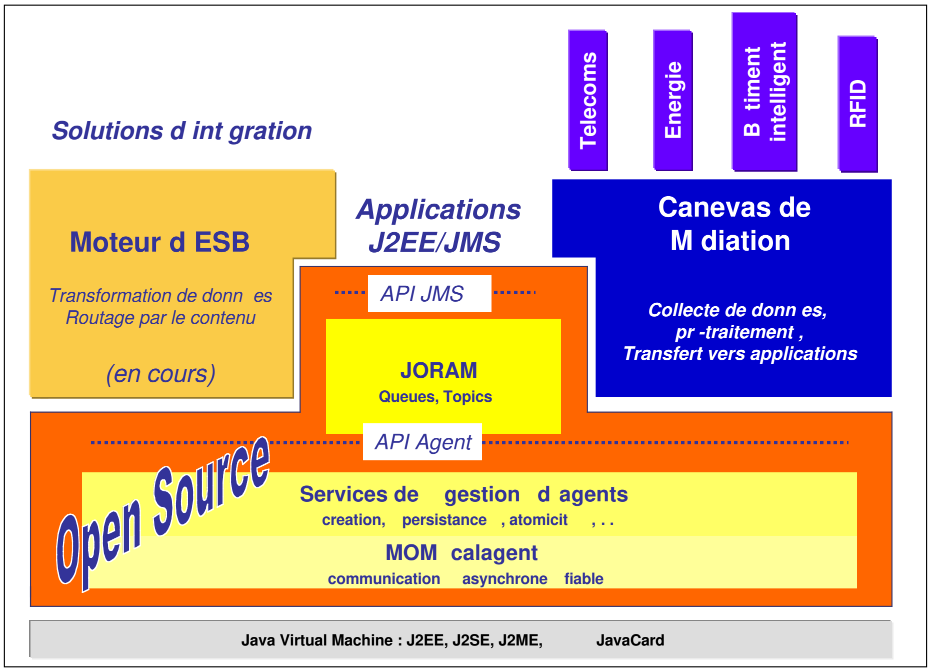

Figure E.19 – Joram et la médiation

Cette approche a été suivie par ScalAgent pour développer une infrastructure d'intégration de données d'entreprise (désignée globalement sous le terme « infrastructure de médiation »). Cette offre regroupe les couches hautes de la figure 19 et comprend les éléments suivants :

- un ensemble de composants de médiation pour la collecte, le traitement et la remontée d'informations d'usage produites par des équipements, services ou applications délocalisés. Ces composants peuvent être personnalisés et assemblés pour décrire des flots de données entre des équipements et des applications métiers. A l'exécution, les composants sont mis en œuvre par des agents exécutés sous le contrôle d'un ensemble de serveurs d'agents.
- Un ensemble d'outils graphiques pour construire, déployer et administrer des solu-

tions d'intégration par personnalisation et assemblage de ces composants. Ces outils s'appuient sur une représentation de type ADL.
– un service de déploiement pour automatiser le processus de déploiement d'une solution d'intégration sur un ensemble de sites.

Cet environnement utilise la même base technologique que JORAM pour gérer la distribution des données et du contrôle. JORAM est également utilisé comme vecteur de communication avec les applications patrimoniales et avec un serveur d'application J2EE où est implantée la logique de traitement des données collectées et remontées par l'infrastructure de médiation. Cette infrastructure de médiation est utilisée pour développer ensuite des solutions verticales adaptées à un usage donné, par exemple : la facturation telécom, la gestion intelligente d'énergie, ou encore la tracabilité à partir de données RFID.

E.7 Références

Documentation en ligne sur JORAM
– Page d'accueil : http ://joram.objectweb.org
– Manuel de l'utilisateur : http ://joram.objectweb.org/doc/index.html
– Wiki JORAM : https ://wiki.objectweb.org/joram/

Publications concernant les bases technologiques de JORAM

Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet and Serge Lacourte. Asynchronous, Hierarchical and Scalable Deployment of Component-Based Applications. In Proceedings of the 2nd International Working Conference on Component Deployment (CD'2004), Edinburgh, Scotland, may 2004

Roland Balter, Luc Belissard and Vivien Quéma. A Scalable and Flexible Operation Support System for Networked Smart Objects. In Proceedings of the 2nd Smart Objects Conference (SOC), Grenoble, France, May 15-17, 2003

Vivien Quéma, Roland Balter, Luc Bellissard, David Féliot, André Freyssinet and Serge Lacourte. Administration and Deployment Tools in a Message-Oriented Middleware. In the 4th ACM/IFIP/USENIX International Middleware Conference (Middleware), Poster session, Rio de Janeiro, Brazil, June 16-20, 2003.

Vivien Quéma, Luc Bellissard and Philippe Laumay. Application-Driven Customization of Message-Oriented Middleware for Consumer Devices. In Proceedings of the Workshop on Software Infrastructures for Component-Based Applications on Consumer Devices, in association with EDOC'02, Lausanne (Switzerland), September 16th, 2002.

L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. Symposium on Reliable Distributed Systems (SRDS'99), Lausanne (Suisse), October 20th-22th, 1999.

R. Balter, L. Bellissard, F. Boyer, M. Riveill and J.Y. Vion-Dury. Architecturing and Configuring Distributed Applications with Olan. In Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), The Lake District (England), September 15th-18th, 1998.

L. Bellissard, N. de Palma and M. Riveill. Dynamic Reconfiguration of Agent-Based Applications. ACM European SIGOPS Workshop, Sintra (Portugal), September 7th-10th 1998.

Documentation relative à JMS

– Page d'accueil JMS : http ://java.sun.com/products/jms/
– Tutorial JMS : http ://java.sun.com/products/jms/tutorial/index.html
– Forum JMS : http ://forum.java.sun.com/forum.jsp ?forum=29
– Introduction à JMS (dans IBM developerwork) : "Enterprise messaging with JMS"
– Nommage et administration (dans IBM developerwork) : "Implementing vendor-independent JMS solutions"
– JMS 1.1 (dans IBM developerwork) : "JMS 1.1 simplifies messaging with unified domains"
– Les architectures applicatives JMS (dans *TheServerSide*) : "JMS Application Architectures"