Chapitre 6

Les modes de communication

L'objectif de ce chapitre est de présenter les différents modes de communication utilisés dans la mise en œuvre d'un SI.
Cette présentation utilise les DP vus précédemment.

1. Introduction et définitions

Une communication entre deux entités du réseau (ou local dans la même machine) est définie par :

- le mode d'échange choisi pour « envoyer » un certain paquet d'information d'un émetteur à un récepteur (ex middleware ou pas, synchrone, asynchrone, broadcast, …) avec
- la possibilité et le moyen de rétroaction (réponse du récepteur. On parle alors de communication bi-directionnelle)
- la nature des informations échangées (binaire, textuel, complexes)
- le protocole (de couche 5) utilisé (FTP, SMTP, NFS, RMI, HTTP, …)

Un **Producteur** est un composant informatique (une classe, un client ou un serveur) qui détient une information (un produit).

Un **Consommateur** est un composant qui doit consommer cette information.

Des exemples d'échange :

- Un producteur envoie cycliquement des informations à un consommateur.
- Un producteur crée des produits en interne et attend que le consommateur vienne chercher les produits.
- Un serveur (le producteur) gère un état (ex: un modèle) en mémoire et un client (le consommateur) demande l'état.
- Un serveur (le producteur) gère un état (ex: un modèle) en mémoire et notifie l'état à un client (le consommateur).
- Un producteur poste des messages dans des boites aux lettres et les consommateurs viennent chercher les messages.
- Un producteur poste des messages dans des boites aux lettres et un postier envoie les lettres aux consommateurs.

Les modes d'échange :

- Communication synchrone : la production d'un "évènement" par un producteur est synchronisée à sa consommation par un consommateur
- Communication asynchrone : la production d'un "évènement" par un producteur n'est pas synchronisée à sa consommation par un consommateur

De plus, il faut prévoir que :

- plusieurs consommateurs peuvent être connectés à un même producteur. Le producteur doit donc "envoyer" l'évènement à tous ses consommateurs ou pas.
- un consommateur peut être connecté à différents producteur.

La communication « synchrone » est basée soit :
- sur l'appel de méthode et requête distante ou
- l'échange d'un message (via des filles de message (ipc par exemple))
pour lequel l'appelant est en attente de réception d'une valeur de retour (même vide).

Pour réaliser cette « attente », il est nécessaire d'utiliser un middleware qui implémente les mécanismes d'attente et de récupération des erreurs (remontée des erreurs par le canal ascendant). De plus, on peut vouloir intégrer des timeouts pour se protéger d'un traitement de l'appelé anormalement long.

Il n'est normalement pas à la charge de l'appelant de mettre en place une telle mécanique. C'est à la charge du Middleware (ou Intergiciel).

Le Midleware doit permettre :
- l'attente de la réception de la valeur de retour

- la récupération du paramètre de retour
- la sérialisation des paramètres
- la propagation de l'exception en cas d'erreur sur la couche de communication

2. Le mode Push synchrone

C'est le cas par exemple du DP Observer/Observable (le "simple" : ex celui de Java)
- la méthode update de l'Observer est une méthode distante appelée par l'Observable
- l'Observer est le consommateur
- l'Observable est le producteur

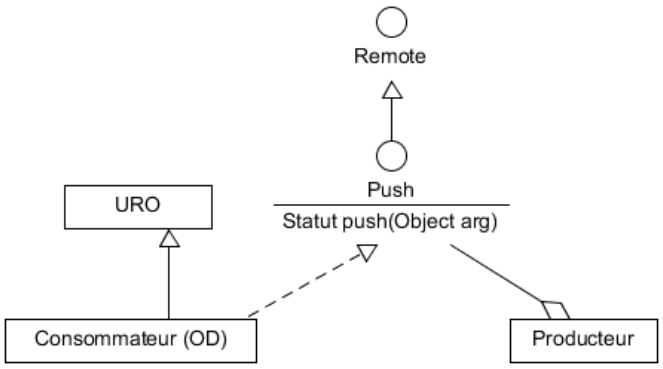
On dit que l'Observable pousse un évènement à l'Observer.

Plusieurs Observer distant peuvent s'abonner à un même Observable.
Un Observer peut s'abonner à plusieurs Observable différents.

Le producteur réalise l'appel de la méthode distante suivante :

consommateur.push(évènement);

Dans le mode Push, le consommateur est le serveur.



2.1. Exemple d'un push : ExempleCh06_01_ModeleMVC_RMI : Le modèle MVC en RMI

a. La problématique

Nous avons vu dans le cadre du cours NSY102-Chapitre-04_LesDesignPatterns, l'exemple Ch04_11_ModeleMVC, l'implémentation de ce patron dans lequel le lien qui relie le Producteur et le Consommateur est un lien d'appels de méthodes locales (update et addObserver, méthodes actionXXX). Il s'agit donc de transformer ces appels locaux sous la forme d'appels distants.

La contrainte est de ne pas modifier l'architecture des classes existantes de l'exemple Ch04_11.

Ainsi ces classes pourront à la demande être utilisées en local et/ou de manière distante.

Ainsi, comme nous allons le démontrer, l'exemple **Ch06_01** inclus l'exemple Ch04_11 puisque l'on pourra exécuter le programme également en local.

Nous faisons l'hypothèse que la communication distante se fait entre les VUE et le CONTROLEUR/MODELE (le modèle et le controleur sont dans la même JVM)

Nous pourrions aussi découpler le contrôleur et le modèle afin qu'ils puissent s'exécuter sur des machines différentes mais il est largement suffisant pour illustrer le cours de se contenter à une communication réseau entre les vues et le controleur/modèle.

b. La solution

La solution peut se résumer avec les choix de conception suivants :

1/ Le contrôleur est transformé en un contrôleur distant qui est un Objet Distribué (= Objet Distant RMI). Les méthodes distantes correspondent au moins aux méthodes du contrôleur.

2/ La classe contrôleur existante reste un singleton qui est toujours utilisé par la vue avec la fonction que la méthode **getInstance**() retourne :

- soit le contrôleur lui-même s'il est utilisé en local **du serveur** (comme avant)
- soit une connexion (lookup) sur le contrôleur distant.

Ainsi, l'utilisation d'un contrôleur distant est transparente pour les vues.

Il est nécessaire quand même de positionner des informations statiques de la classe permettant de déterminer le mode de fonctionnement du contrôleur (local ou distant). Ces informations contiennent au moins le hostname et le port de l'adaptateur du contrôleur distant.

3/ L'abonnement par une vue est réalisé par le contrôleur distant

4/ C'est lui qui doit mettre en place le mécanisme permettant que la notification se fasse de manière distante.

Ainsi, la communication entre l'Observable et l'Observer se fait par l'appel d'une méthode distante. L'Observer est transformé en un observer distant. Le **stub** de cet observer distant est transmis au contrôleur distant à travers l'abonnement afin que un observer dédié appel la méthode distante du stub pour réaliser la notification.

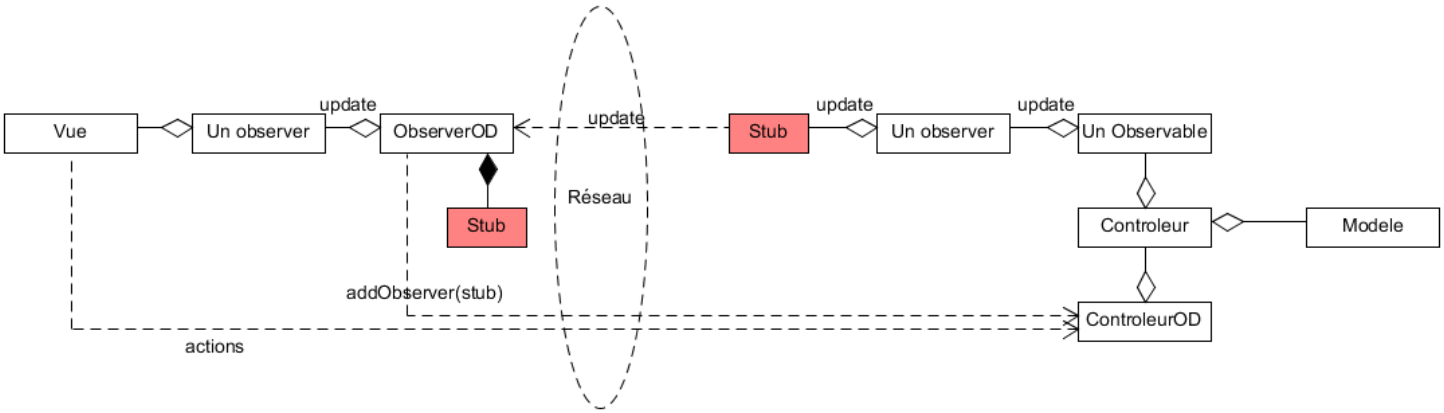
Remarque :

On peut également ne pas passer le stub mais passer le host et le port de l'adaptateur de l'observer distant afin que l'observable fasse un lookup à chaque notification. Cela entraine que chacun de ces observers distants s'enregistre dans un adaptateur de leur machine respectif.

Cela peut être un principe de robustesse par rapport à une délocalisation temporaire de la vue.

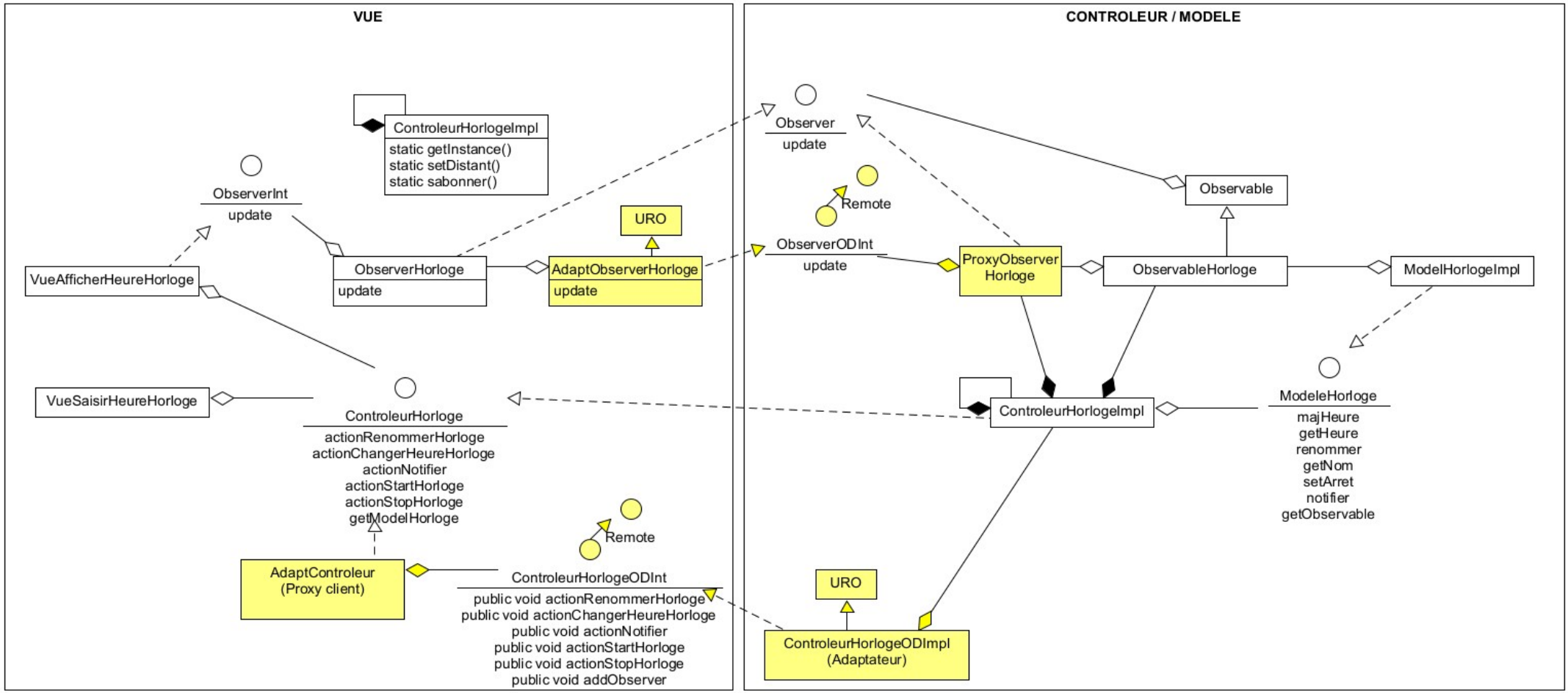
Cette remarque de robustesse est valable pour la communication entre les vues et le controleur/modele : chaque appel à une action réalisant un lookup.

On peut schématiser ce fonctionnement ainsi :



c. Diagrammes de classe

L'architecture des classes de cet exemple est la suivante :
(modification)

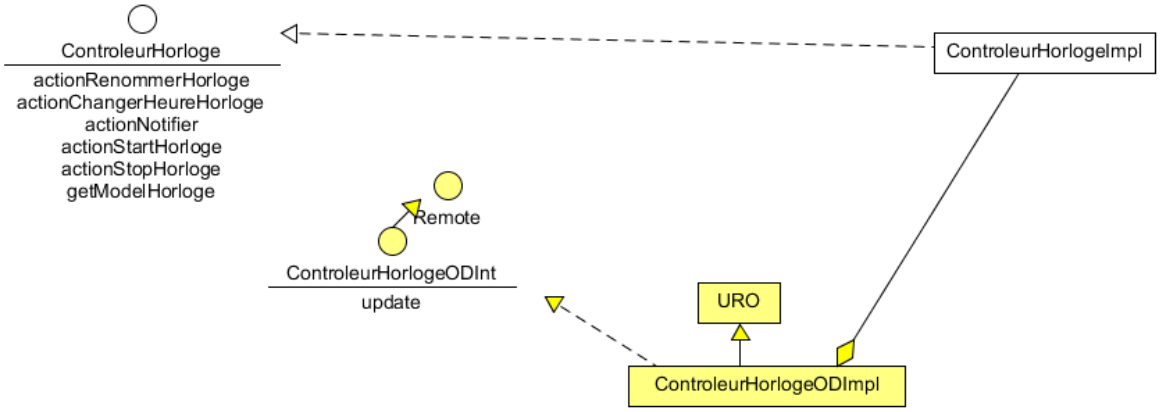


En jaune, les classes et interface ajoutées par rapport au schéma de l'exemple Ch04_11.

d. La communication synchrone distante entre la vue et le contrôleur

Intéressons-nous plus particulièrement à l'architecture de communication entre une vue et le contrôleur distant.

Pour garder la compatibilité d'un comportement locale avec celui d'un comportement distant, nous ne changeons pas la classe du contrôleur mais créons un objet distribué sous la forme du **DP Adaptateur** permettant d'utiliser le contrôleur d'une manière distante.



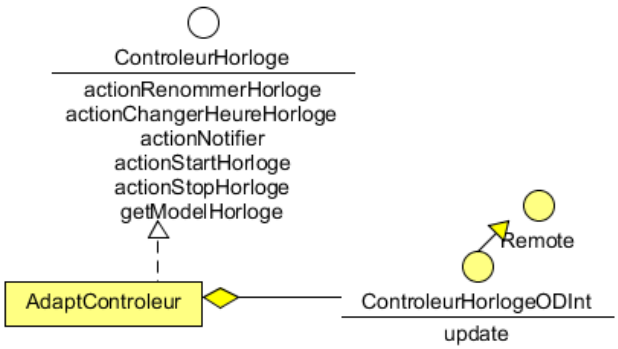
La classe **ControleurHorlogeODIImpl** est un Objet distribué qui adapte le contrôleur à l'interface distante **ControleurHorlogeODInt**

La méthode **main** de cette classe crée cet objet distribué et l'enregistre dans l'annuaire sous le nom CONTROLEUR_HORLOGE. Ce main est le programme principal du Service.

Ainsi l'interface RMI **ControleurHorlogeODInt** permet d'utiliser à distance le contrôleur. Toutes les méthodes de l'interface **ControleurHorloge** sont implémentées dans cette interface.

Afin de rendre transparent, pour les vues du contrôleur, l'appel à ces méthodes distantes, nous créons un **Adaptateur de conversion : AdaptControleur**.

Cet adaptateur de conversion convertit l'interface **ControleurHorlogeODInt en ControleurHorloge.**



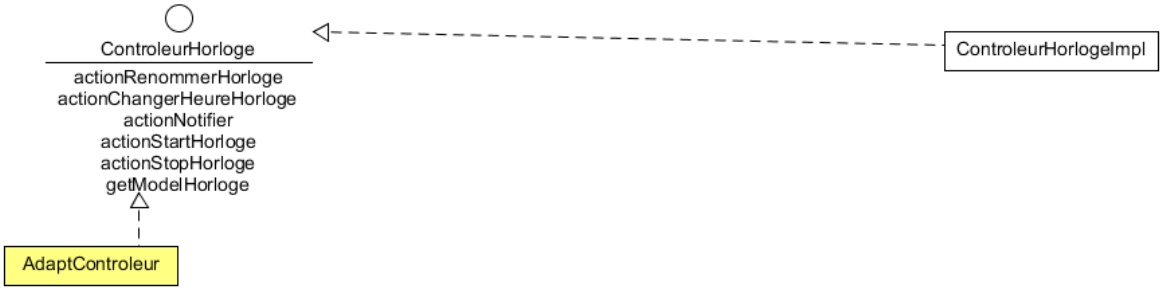
Cet adaptateur est utilisé par les vues. Il est créé par la méthode **getInstance** du controleur (singleton) quand le controleur est utilisé de manière distante.

La méthode **public static void setDistant(String …)** précise que le controleur est utilisé en mode distant. Cette méthode précise en argument le hostname et le port RMI du controleur distant.

```
static public ControleurHorloge getInstance()
{
    if (distant)
        {
            try{
                String addr = "rmi://"+host+":"+port+"/CONTROLEUR_HORLOGE";
                ControleurHorlogeODInt  cd = (ControleurHorlogeODInt)(
                                                Naming.lookup(addr));
                return new AdaptControleur(cd);
            }catch(Exception ex){System.out.println(ex+"\nImpossible lookup
                                                            sur controleur: "
                                                    +host+" "+port);};
        }

    if (modelHorlogeSingleton==null)
        {
            modelHorlogeSingleton  =
              new ModelHorlogeImpl("TOULOUSE", new ObservableHorloge() );
            controleurHorlogeSingleton =
              new ControleurHorlogeImpl(modelHorlogeSingleton);
        }
    return controleurHorlogeSingleton;
}
```

Remarque : L'adaptateur AdaptControleur est aussi un proxy de communication du Controleur :



e. La communication synchrone distante entre la vue et le modèle

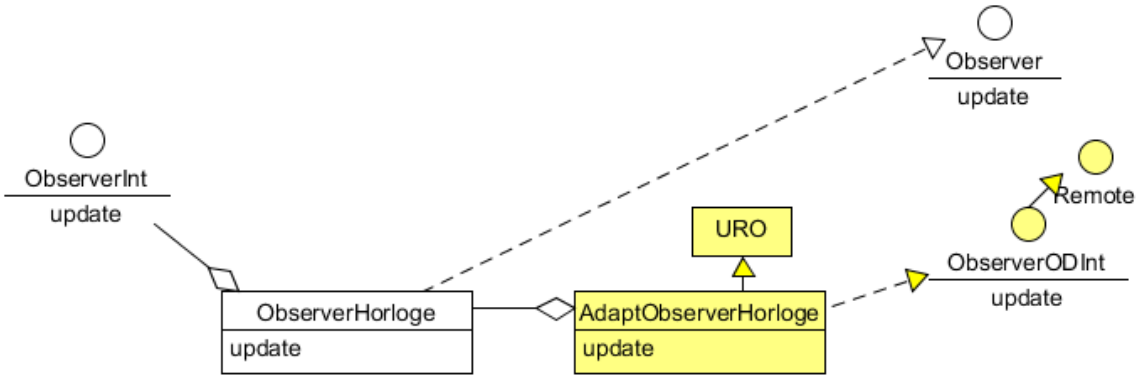
Intéressons-nous maintenant à l'architecture de l'Observer/Observable qui doit maintenant se faire via des appels distants.

Il s'agit de ne pas modifier les classes de l'exemple Ch04_11.

Côté client :

Le principe est de transformer l'observer **ObserverHorloge** en un Objet Distribué **AdaptObserverHorloge** qui devra recevoir les notifications du serveur via l'appel d'une méthode distante (update). Il relaie ensuite la notification à l'observer de la vue). Cette classe est un DP Adaptateur de conversion de l'interface Observer en ObserverODInt.

Pour cela on crée l'OD en un Adaptateur de l'objet cible :

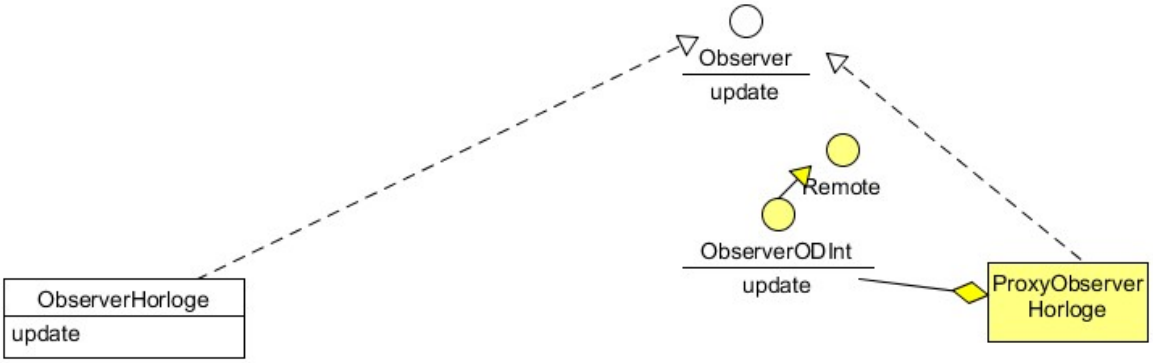


Il faut alors transmettre au serveur le stub de cet objet distribué afin qu'il appelle la méthode distante update.

Cela se fait tout simplement par l'abonnement : **addObserver(stub)** qui est une méthode distante du controleur distant.

Côté serveur :

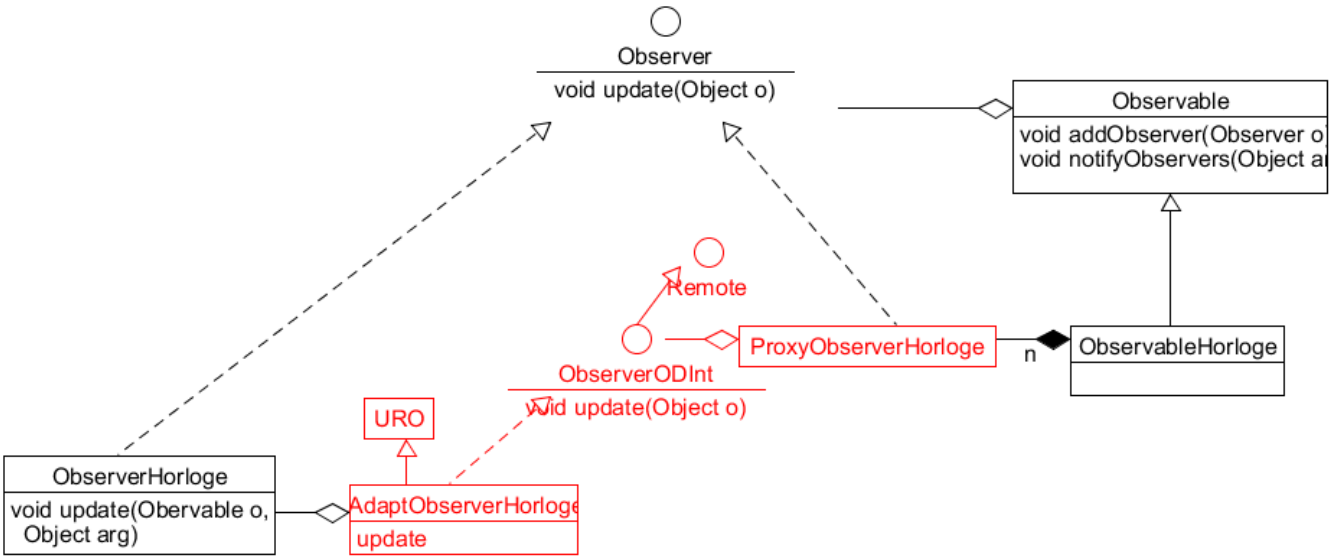
Il faut donc maintenant que le stub s'abonne à l'Observable. Pour cela on crée un DP Proxy **ProxyObserverHorloge**



C'est ce proxy qui s'abonne à l'Observable.

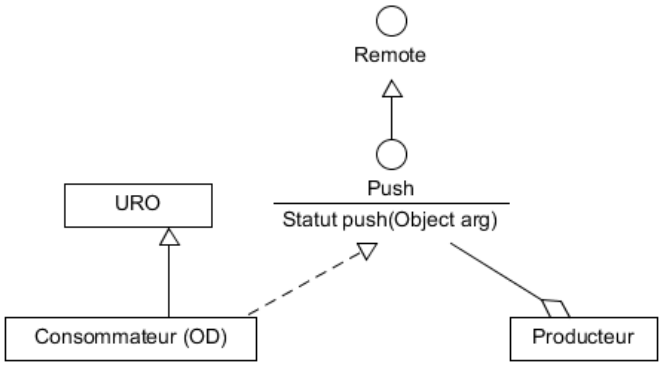
Ainsi à chaque notification elle appelle la méthode update du stub **ObserverODInt**.

Synthèse :
Le schéma suivant représente en synthèse le DP d'un Observer/Observable distant :

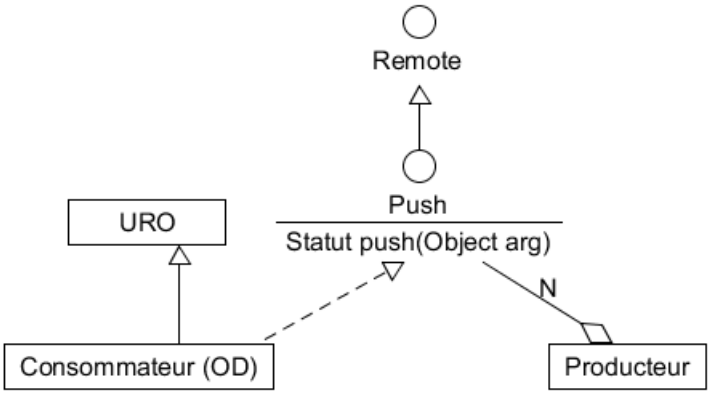


Ici, en rouge, on reconnait le DP Push synchrone distant à faible couplage.

Si on compare ce DP à celui décrit en début de ce chapitre :



La différence est dans le degré de couplage entre le Producteur et les consommateurs. On devrait écrire :



Ici le couplage (N) entre le Producteur et ses consommateurs est un couplage fort. Le Producteur connaît exactement les Consommateurs à qui il doit pousser une information.

Voir sur le site http://jacques.laforgue.free.fr l'exemple
ExempleCh06_01_ModeleMVC_RMI

2.2. Exercice

Voir sur le site http://jacques.laforgue.free.fr dans les Exercices :
Exercice02_ConnecteurPush

3. Le mode Pull synchrone

Le mode pull synchrone se fait en inversant l'appel de la méthode distante.

Ce n'est plus le producteur qui appelle une méthode distante du consommateur pour lui pousser un évènement mais c'est le consommateur qui appelle une méthode distante du producteur pour obtenir un évènement.

Le consommateur réalise l'appel de la méthode distante suivante :

évènement = producteur.pull();

Dans le mode Pull, le producteur est le serveur.

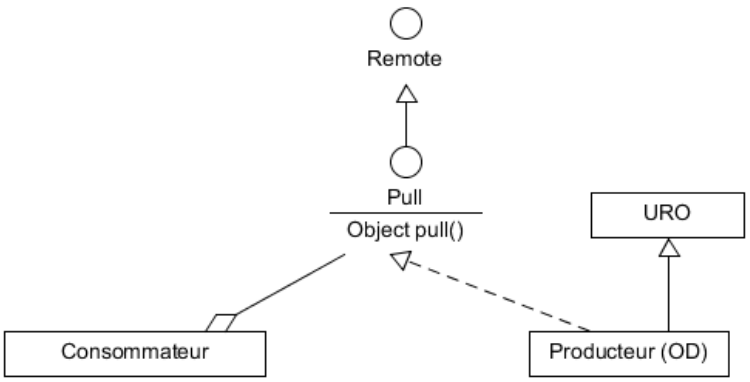
Dans ce cas la méthode pull est "synchrone".

Le consommateur attend le retour d'appel de la méthode puisqu'il attend en retour l'évènement.

Cela signifie que si plusieurs consommateurs utilisent la méthode distante en même temps, tant qu'un consommateur utilise la méthode, les autres sont en attente si la méthode est synchrone sinon non.

C'est le midlleware (RMI) et/ou le langage qui met en place ce mécanisme de file d'attente des appels de la méthode distante.

En RMI cela se fait en ajoutant devant la méthode le mot "synchronize".



Il existe 2 cas :

- le cas où l'évènement est un état unique du producteur (qu'il met à jour à son rythme)
- le cas où les évènements s'accumulent dans une file.

Le 1er cas ne pose pas de problème particulier.

Remarque : conséquence le consommateur peut récupérer le même évènement consécutivement ou ne rien récupérer (valeur null).

Par contre le 2ème cas, nécessite de mettre en place une politique de la consommation des évènements car les consommateurs ne consomment pas les évènements au même rythme.

Dans ce cas, on a une production des évènements qui est découplée de leurs consommations. On est donc dans le cas d'une communication asynchrone : le mode Pull asynchrone. (On est proche du modèle Push/Pull qui est vu plus bas : le producteur pousse ces évènements dans la file. Les consommateurs tirent les évènements de la file.)

4. Le mode Pull asynchrone

La problématique :

On veut que la consommation des évènements par les consommateurs soit asynchrone par rapport à leurs productions et nous voulons également que les consommateurs tirent, indépendamment de leurs rythmes, tous les évènements produits. Sachant que le producteur gère 1 seul groupe physique d'évènements.

La solution :

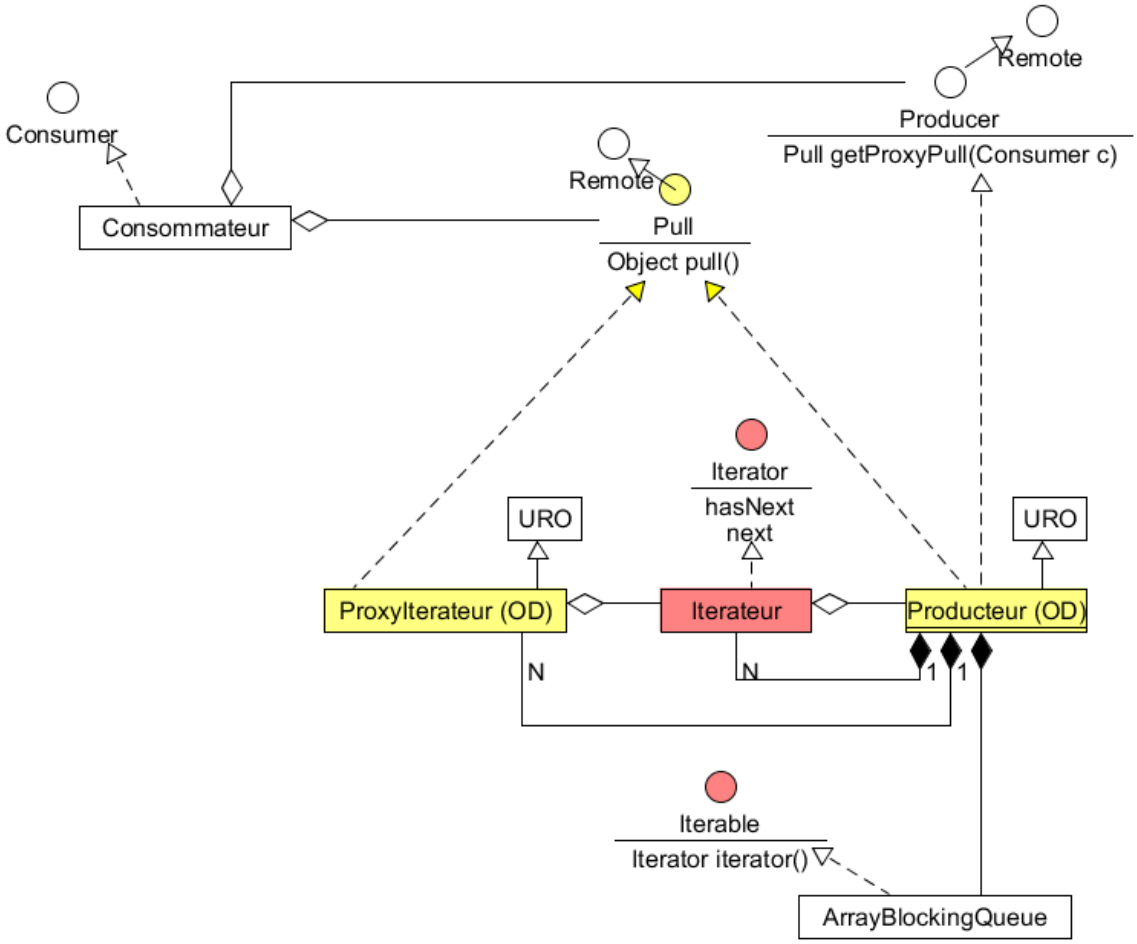
La solution consiste à donner un Itérateur (DP Iterator) à chacun des consommateurs.

Chaque consommateur se "déclare" donc au producteur.

Chacun des itérateurs maintient donc la séquentialité de la consommation des évènements pour chacun des consommateurs.

Remarque : Les 2 cas (évènement unique ou multiple) nécessitent de résoudre le problème du "lecteur/écrivain" de la gestion de l'évènement (état) ou de la file d'évènement. Cela est pris en charge par la classe ArrayBlockingQueue (il est intéressant d'aller voir dans l'API Java l'interface BlockingQueue).

La solution peut être la suivante :



Le consommateur utilise la méthode Pull getProxyPull(Consumer c) pour obtenir le **stub** de ProxyIterateur qui implémente la méthode pull qui appelle la méthode next de l'itérateur.

Cette méthode crée à chaque appel l'instance de ProxyIterateur qui crée un nouvel itérateur (en appelant la méthode iterator de ArrayBlockingQueue).

Dans ce schéma on reconnaît :

- un DP Proxy : la classe ProxyIterateur est le proxy de Producteur
- un DP Iterator
- un DP OD par héritage
- un DP pull
- un DP interface ☺

Remarque : si les consommateurs suppriment de la file l'élément tiré alors on a 1 seul iterateur, sinon on a autant d'iterateur que de consommateur.

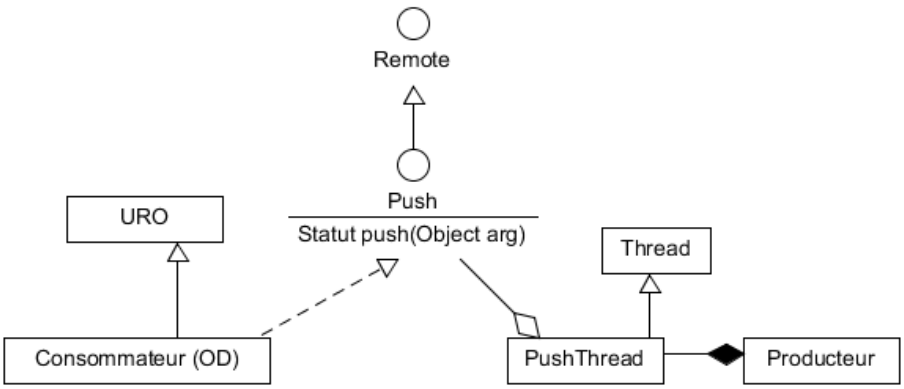
Dans le cas d'autant d'itérateur, il faut que les consommateurs se déclarent tous au producteur pour que l'élément soit supprimé quand tous les consommateurs ont consommé l'élément (pour des raisons de performance).

5. Le mode Push asynchrone

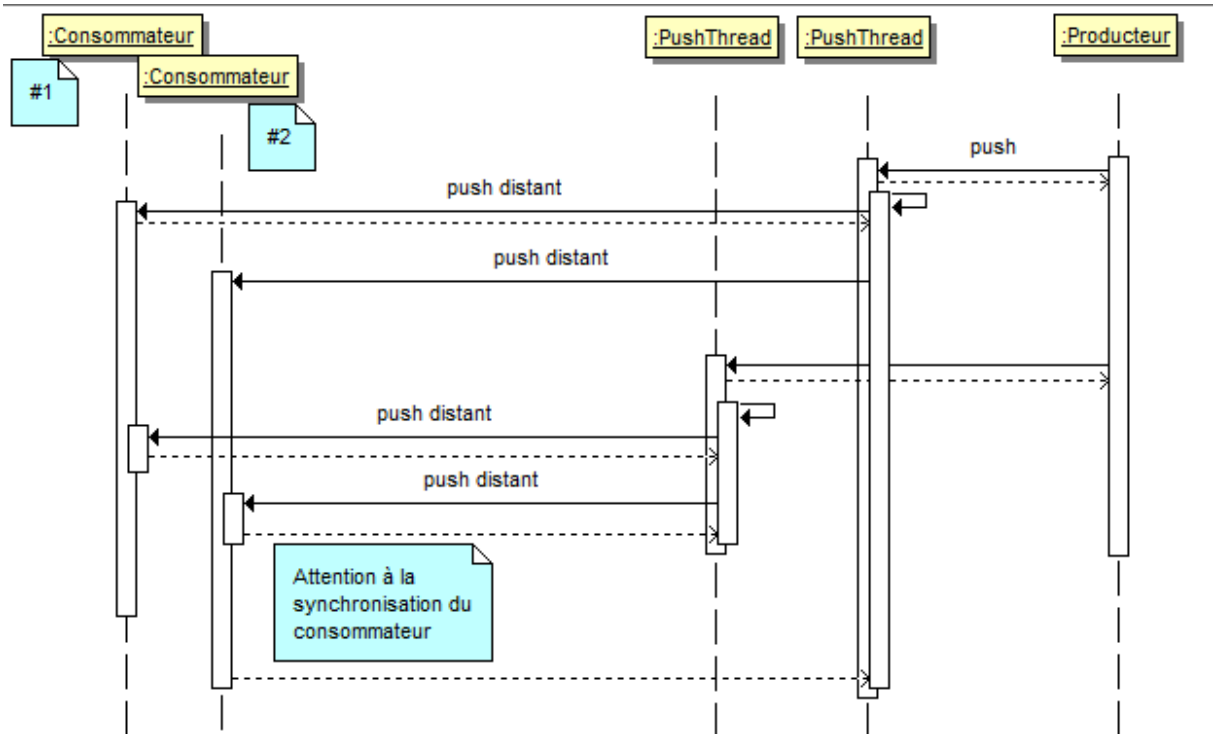
Comme cela a été dit en introduction, la communication asynchrone consiste à ne pas attendre le retour de l'appel de la méthode push.

Il suffit de faire l'appel de la méthode push dans un thread.

Remarque : Le protocole RMI ne permet pas de réaliser l'appel d'une méthode distante de manière asynchrone. En CORBA cela est réalisable (oneway)



Le diagramme de séquence :



6. Un canal d'évènement (un intermédiaire)

6.1. Définitions

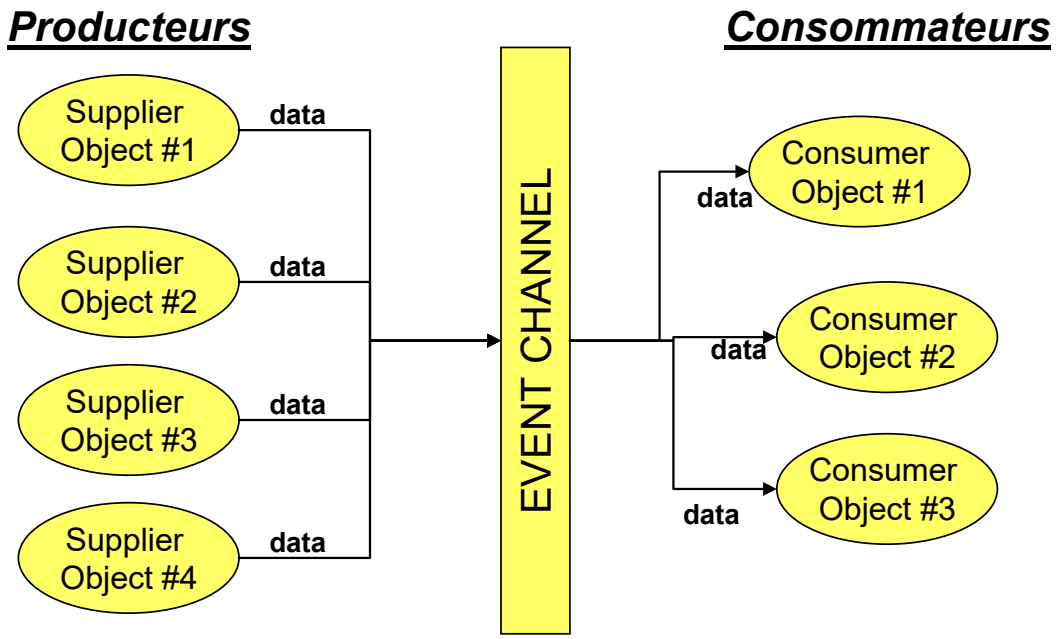
Dans les architectures précédentes, il y a un couplage fort entre les consommateurs et les producteurs : les producteurs connaissent l'existence des consommateurs.

Par exemple, dans le mode Pull, les consommateurs doivent connaître le producteur afin de se connecter pour utiliser les méthodes distantes (ex en RMI, le lookup sur le producteur).

Ainsi les principes du canal de communication sont :

- les producteurs se connectent au canal sans connaître l'existence des consommateurs.
- les consommateurs se connectent au canal sans connaître l'existence des producteurs
- le canal assure la transmission des évènements entre les producteurs et les consommateurs (en mode push ou pull)

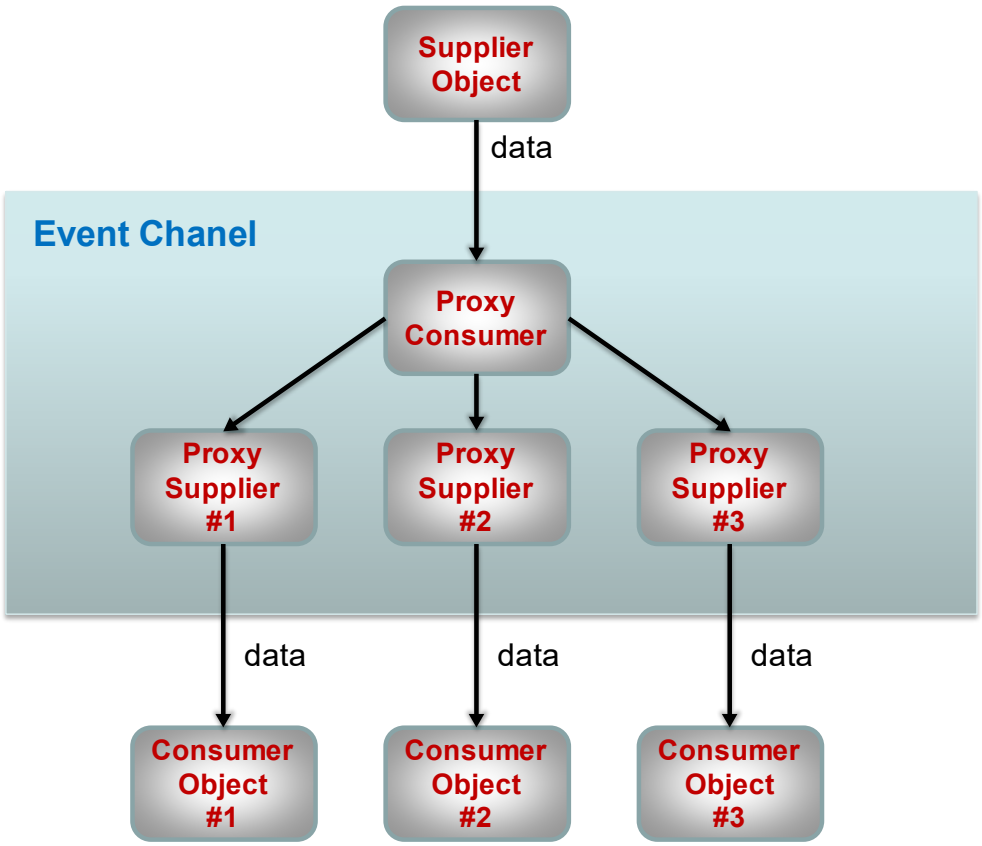
Cet intermédiaire est un composant du Middleware.



Les producteurs et les consommateurs sont découplés :

- utilisation de "proxy" (intermédiaire)
- les proxy sont gérés par le canal d'évènements
- un producteur voit les consommateurs à travers un PROXY
- un consommateur voit le producteur à travers un PROXY

Schéma pour 1 canal d'évènement :



L'entité « Producteur » (Supplier) transmet une donnée (soit il la pousse, soit on lui la tire) au canal d'évènement qui est une entité à part entière (intermédiaire).

Le producteur ne sait pas le nombre et la nature des consommateurs « connectés » au canal d'évènement.

Dans un SI, il est bien sur possible de créer autant de canaux d'évènement que nécessaire.

6.2. Le mode « push »

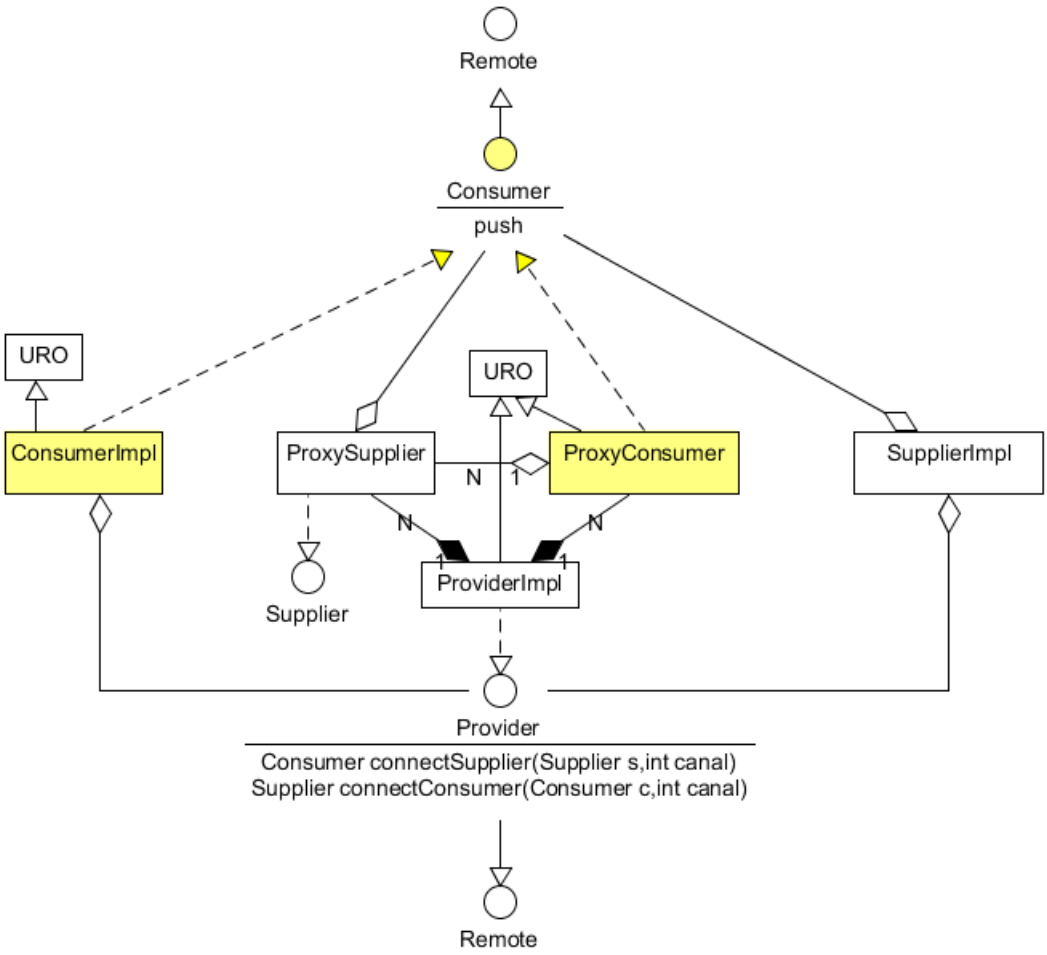
Les principes :

- le producteur contrôle le flot de données en étant à l'initiative : il pousse la donnée "aux consommateurs" en utilisant le proxy
- le canal d'évènement « reçoit » donc la donnée
- il pousse à son tour la donnée à chacun des consommateurs

Le producteur n'est pas directement en contact avec chacun des consommateurs mais avec une seule entité (ProxyConsumer) qui sert donc d'interface avec tous les consommateurs. Le producteur utilise les méthodes distantes de l'interface **Consumer** qui contient la méthode **push.**

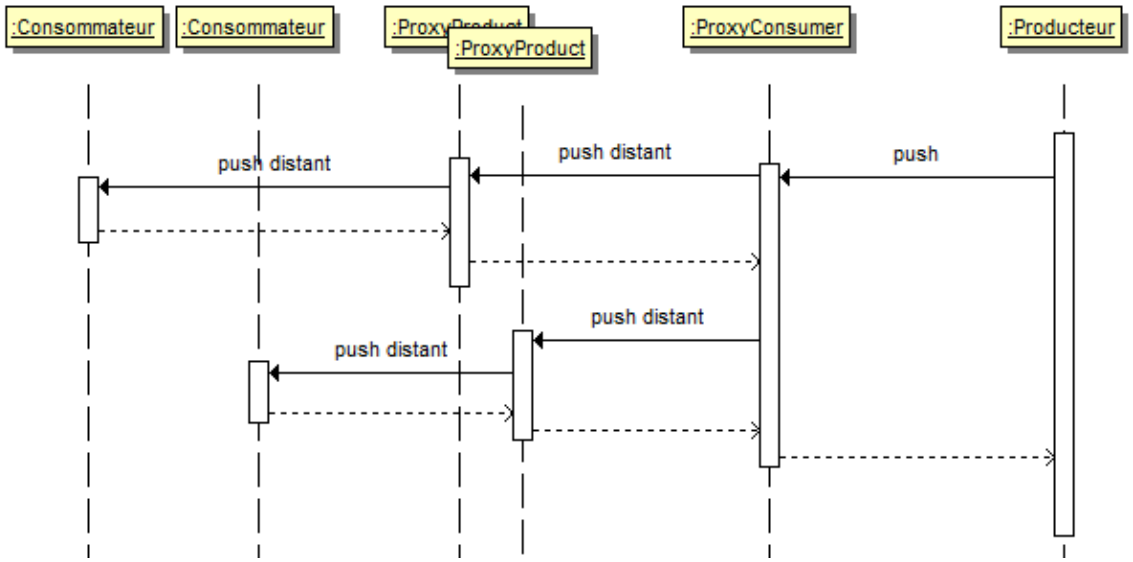
Le producteur se déclare au canal d'évènement.

Chaque consommateur se connecte au canal d'évènement qui crée pour chacun d'eux un producteur spécifique (ProxySupplier) dont le rôle est d'appeler la méthode **push** implémentée par le consommateur.



On note ici en jaune que ProxyConsumer est bien un proxy de ConsumerImpl car ils implémentent tous deux l'interface Consumer

En diagramme de séquence :



6.3. Le mode « pull »

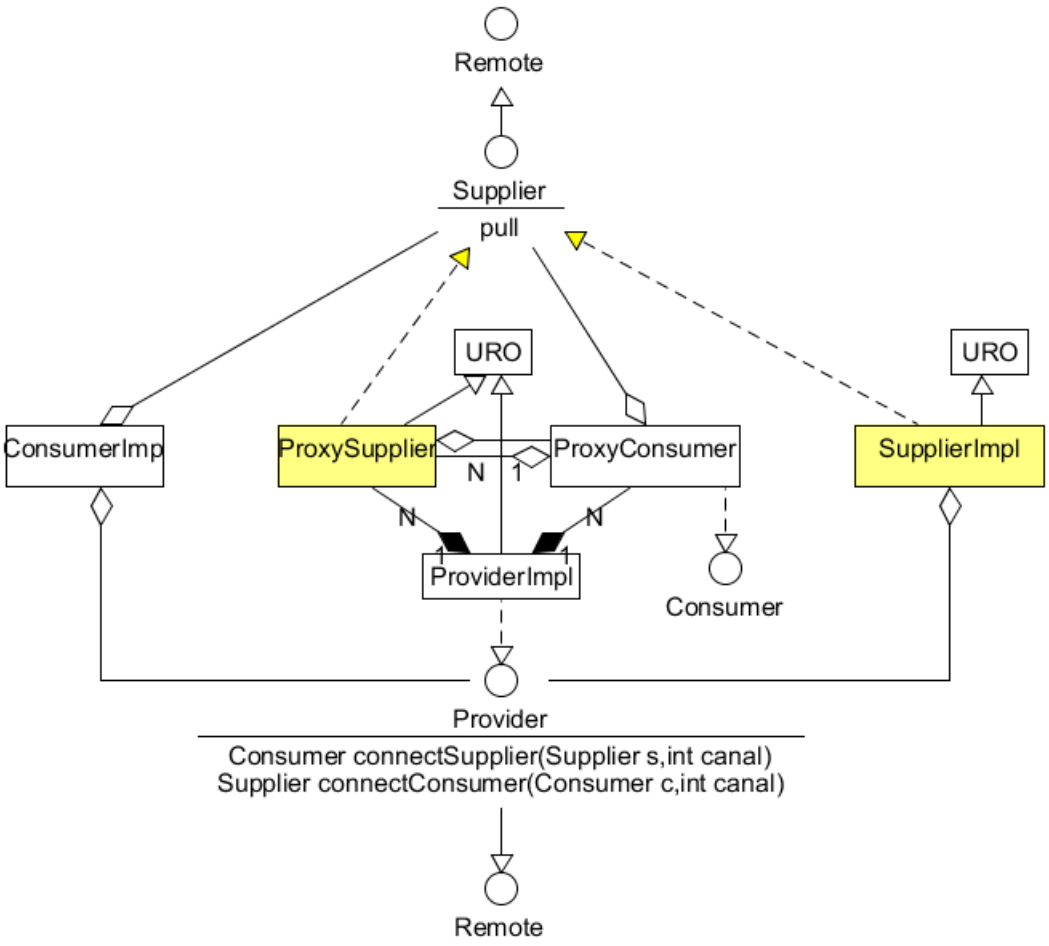
Les principes :

- les consommateurs contrôlent le flot de données en étant à l'initiative : ils tirent la donnée au producteur
- le consommateur tire la donnée à l'intermédiaire
- l'intermédiaire (canal d'évènement) tire la donnée au producteur

Les consommateurs ne sont pas directement en contact avec le producteur mais avec, pour chacun, une entité (ProxySupplier) qui sert donc d'interface avec le producteur. Le consommateur utilise les méthodes distantes de l'interface **Supplier** qui contient la méthode **pull**.

Chaque consommateur se déclare au canal d'évènement.

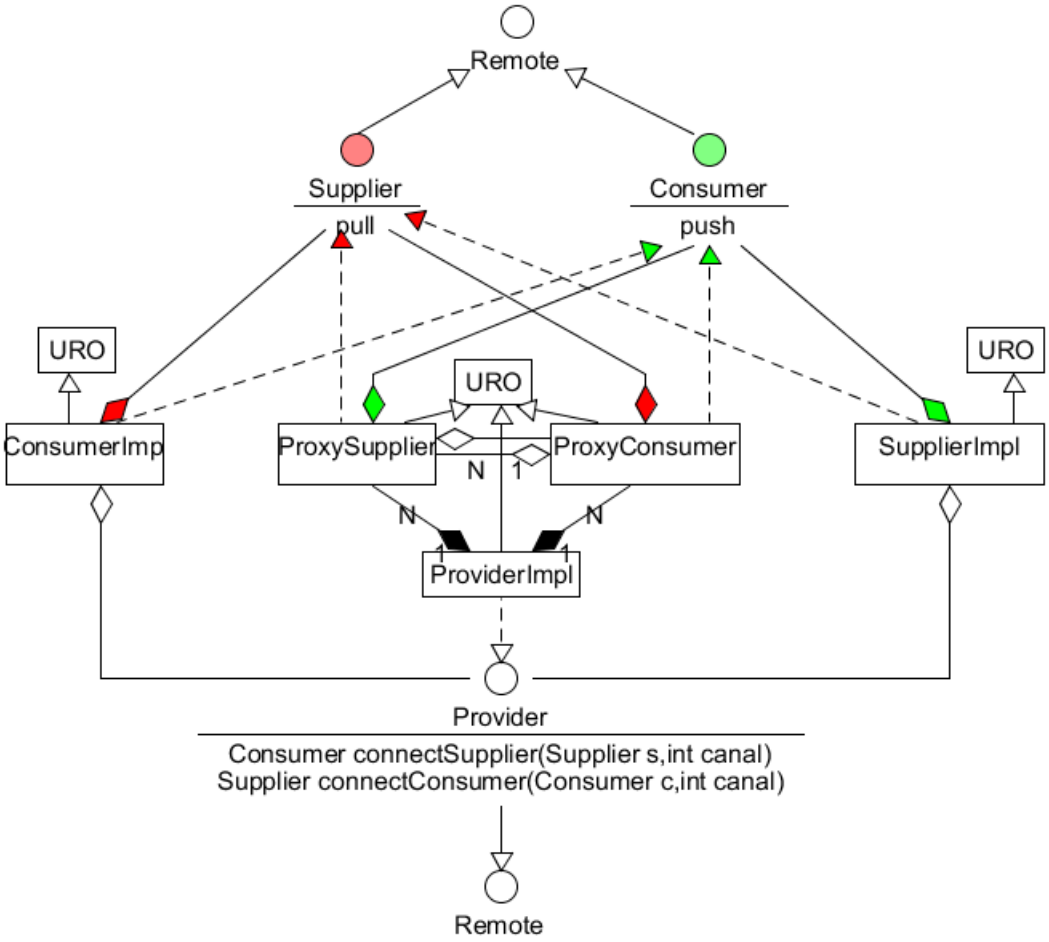
Le producteur se déclare au canal d'évènement qui crée pour lui un consommateur spécifique (ProxyConsumer) dont le rôle est d'appeler la méthode **pull** implémentée par le producteur.



On note ici en jaune que ProxySupplier est bien un proxy de SupplierImpl car ils implémentent tous deux l'interface Supplier

6.4. Le modèle complet

Si on réalise le diagramme complet, on obtient le Design Pattern suivant :



Les mêmes composants servent à la fois pour faire soit du push, soit du pull.

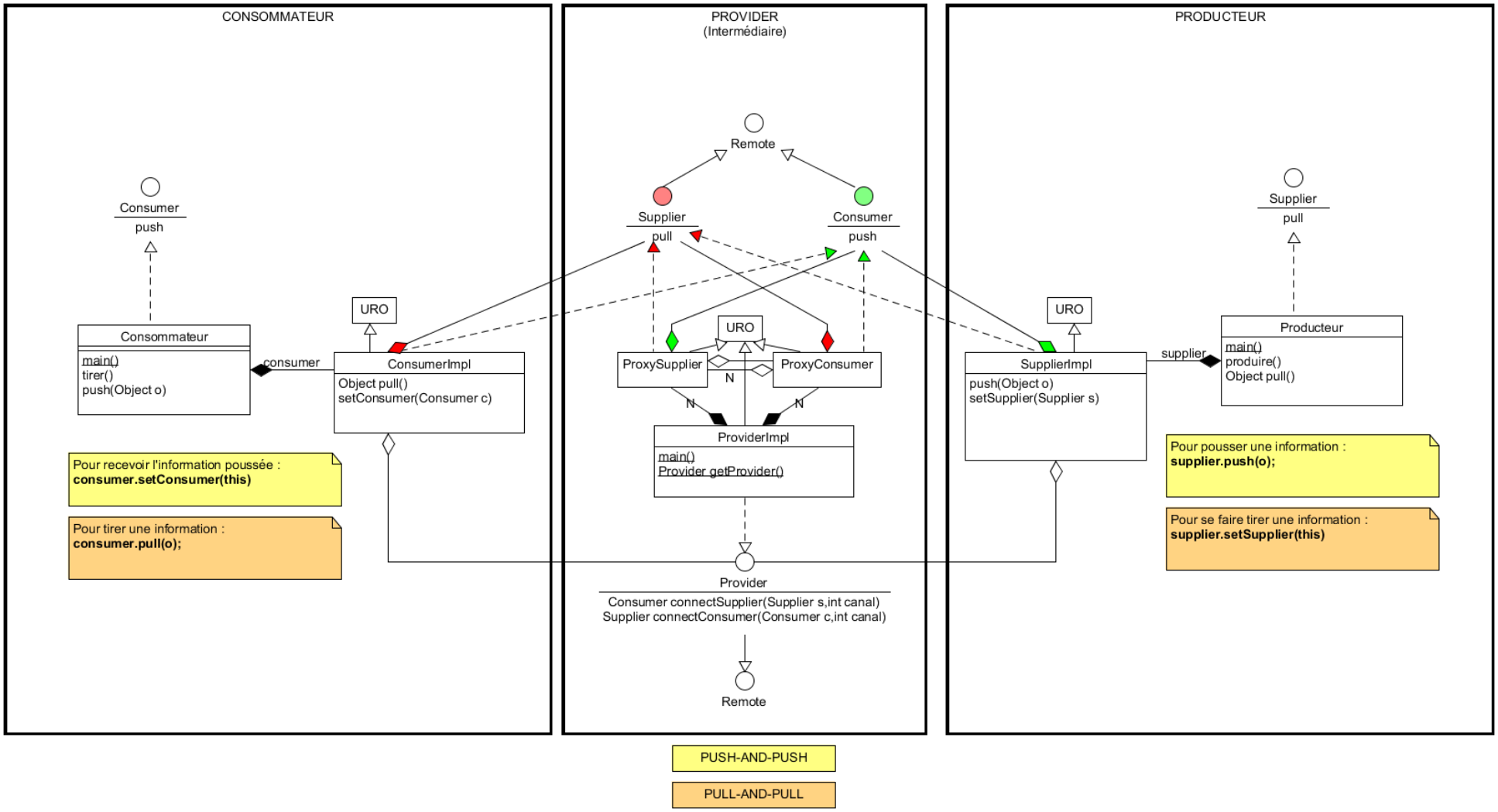
On peut comprendre que ces deux types de communication, push et pull, sont synchrones. Tout dépend si la méthode push ou pull se fait dans un thread.

Différence de mode de communication dans les 2 modèles : initiative du producteur (push), initiative du consommateur (pull).

6.5. Exemple d'implémentation du canal d'évènement

Voir sur le site http://jacques.laforgue.free.fr dans les Exemples :
ExempleCh06_02_Provider

Cet exemple est le codage de ce DP.


CONSOMMATEUR
PROVIDER
(Intermédiaire)
PRODUCTEUR
Remote
Consumer
push
Supplier
pull
Consumer
push
Supplier
pull
URO
URO
URO
Consommateur
main()
tirer()
push(Object o)
consumer
ConsumerImpl
Object pull()
setConsumer(Consumer c)
ProxySupplier
ProxyConsumer
N
N
N
ProviderImpl
main()
Provider getProvider()
SupplierImpl
push(Object o)
setSupplier(Supplier s)
supplier
Producteur
main()
produire()
Object pull()
Pour recevoir l'information poussée :
consumer.setConsumer(this)
Pour tirer une information :
consumer.pull(o);
Pour pousser une information :
supplier.push(o);
Pour se faire tirer une information :
supplier.setSupplier(this)
Provider
Consumer connectSupplier(Supplier s,int canal)
Supplier connectConsumer(Consumer c,int canal)
Remote
PUSH-AND-PUSH
PULL-AND-PULL

Commentaire de cet exemple en cours.

Cette implémentation ne gère pas une file des données fabriquées par le Producteur.
Les deux modes de communication implémentés dans cet exemple sont :

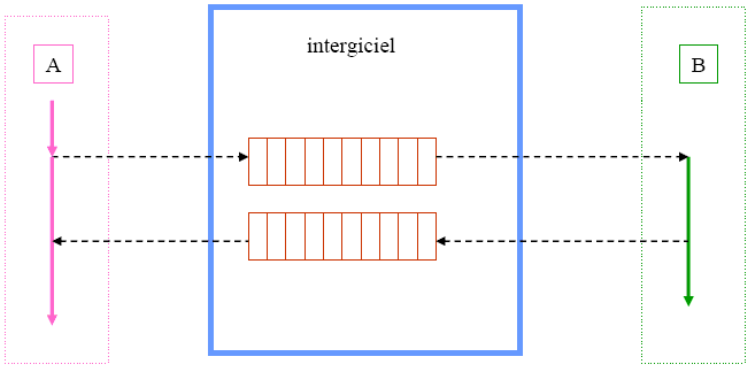
- le mode push-and-push
- le mode pull-and-pull

Si on implémente une file de message dans le Provider avec autant d'itérateur que de consommateur, on pourra réaliser le mode **push-and-pull asynchrone**.

Exercice à faire : faire le diagramme UML de cette implémentation.

6.6. Le mode push-and-pull et push-and-push asynchrone

On a vu que ce modèle de DP peut être utilisé soit en push, soit en pull mais il peut être utilisé aussi en **push-and-pull** qui est par définition asynchrone avec l'utilisation de files de message implémentées dans l'intermédiaire.



push-and-pull (le mode "Queue") :

- le producteur pousse les données dans l'intermédiaire qui gère une file
- le consommateur tire les données à son rythme

push-and-push (le mode "Topic") :

- le producteur pousse les données dans l'intermédiaire qui gère une file
- l'intermédiaire pousse, **à son rythme**, les données **vers tous** les consommateurs.

7. Les MOM

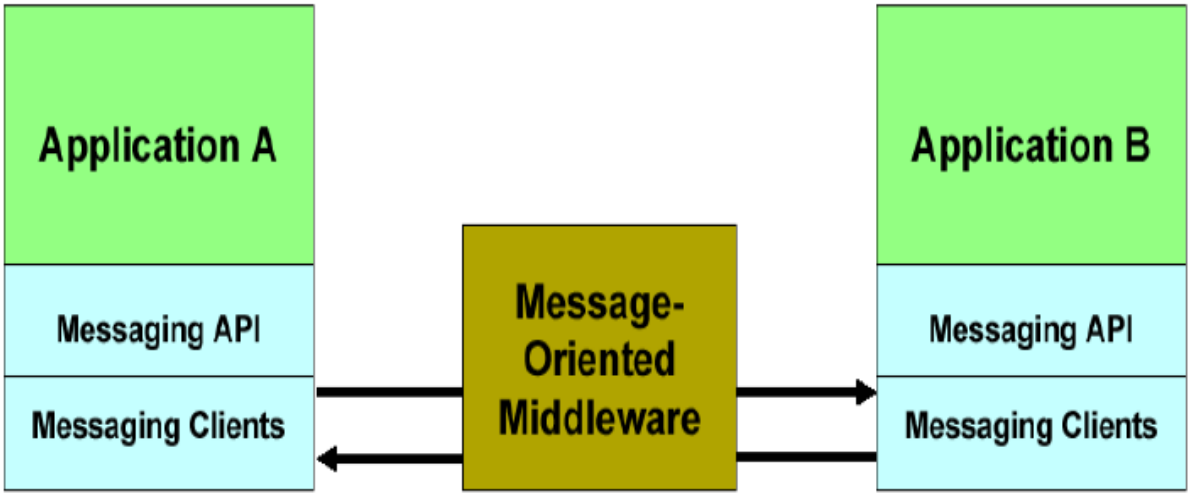
On retrouve les canaux d'évènement dans les MOM (Message Oriented Middleware) : un middleware orienté message.

MOM est une "spécification" (ou principes) implémentée dans un logiciel comme par exemple JMS une API de J2EE (Java Messaging System).

Un tel middleware est une boite à outil, utilisé dans la couche métier ou intranet, et aussi internet à travers les WebServices, pour pouvoir faire communiquer les composants métiers (ou service WS) entre eux. On parle de services d'échanges entre les applications.

Le MOM est un tier supplémentaire. Il diffère aux appels de méthodes distantes (comme RMI ou CORBA) par deux principes forts :

- un composant actif du MOM sert d'intermédiaire entre les composants pour les faire communiquer
- le MOM permet de réaliser une communication asynchrone c'est-à-dire sans que les applications destinatrices soient connectées lors de l'émission des messages. Cela a une conséquence, l'intermédiaire contient une ou plusieurs files de message contenant les messages produits.



Les producteurs se connectent à un canal d'évènement. Il produit ces évènements à son rythme qui sont stockés dans la file.
Ensuite, les consommateurs se connectent au canal d'évènement pour consommer les évènements.

Remarque : Une même application peut être à la fois un producteur dans un canal et consommateur dans un autre.

Les grandes fonctions d'un MOM :

- Ouvrir des canaux de communication entre les composants
- Fournir une API permettant de créer le Message (Une enveloppe gérée par le MOM permettant le transport de toute information, indépendant du format transporté) et permettant de produire et consommer un message
- Assurer la persistance des messages dans le temps jusqu'à leur consommation
- Permettre une communication point à point entre deux composants
- Permettre une communication multi-points (Producteurs/Consommateurs)
- Gérer les priorités dans la délivrance des messages

- Gérer des dates d'expiration de message
- Différer des messages dans le temps
- Gérer des services d'adressage et de routages des messages
- Gérer des systèmes d'alerte lorsqu'un message se présente ou selon un volume de messages atteignant un seuil trop important.
- Gérer les flux de messages, les congestions et goulets d'étranglement. Liste de fonctionnalités non exhaustive.
- ... etc ...

7.1. Les modes de communication dans un MOM

Il existe deux modes de consommation des évènements :

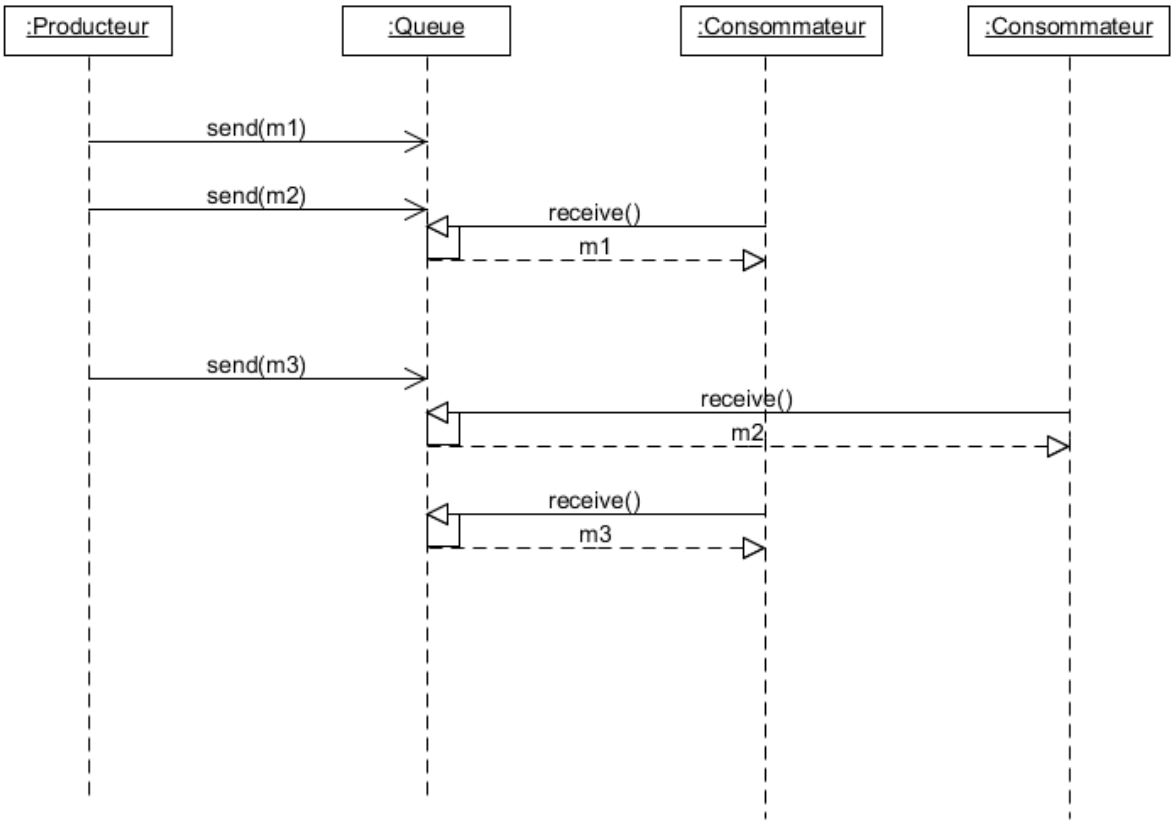
- mode "Queue"
- mode "Topic"

La transmission des données entre le producteur et les consommateurs est asynchrone.

7.1.1. Le mode "Queue"

Dans ce mode (classique de la "file de message") :

- le producteur dépose un message dans la file (géré en FIFO)
- le consommateur prend le message (il est supprimé de la file, éventuellement)

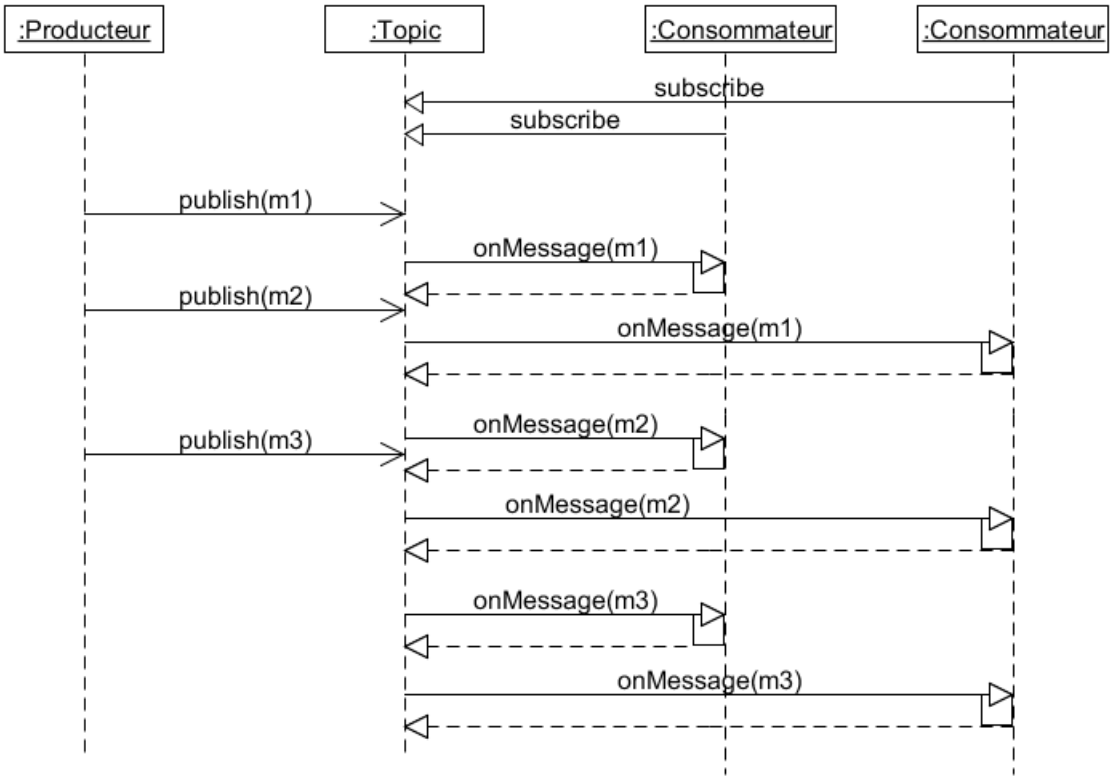


On est dans un modèle **Push-Pull** asynchrone.

7.1.2. Le mode "Topic"

Dans ce mode, les consommateurs se déclarent au préalable au canal d'évènement.
Ils s'abonnent en créant un listener qui prend en entrée un objet qui implémente la méthode onMessage qui sera appelée par l'intermédiaire pour lui envoyer le message.

Ce mode de communication assure que tous les messages publiés seront transmis à tous les consommateurs.



On est dans un modèle **Push-Push** asynchrone
➔ Push/asynchrone – Push/Synchrone (sans Thread)

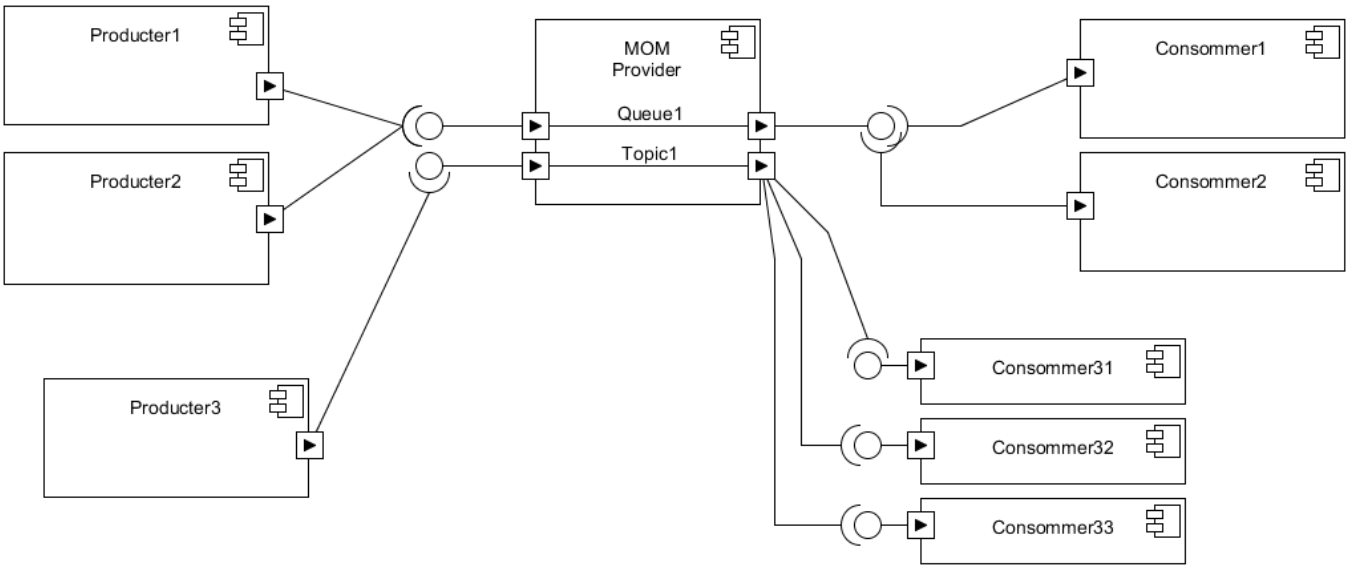
Autre principe :
Push/asynchrone – Push/Asynchrone (avec Thread)

~~Faire le schéma~~

On est asynchrone car le producteur n'est pas en attente de l'envoi des messages à tous les consommateurs abonnés pour émettre un nouveau message.

7.2. La représentation dans une Configuration Architecturale

Dans une Configuration Architecturale, on peut représenter les composants d'un MOM ainsi :

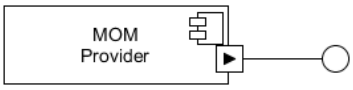


Il faut identifier dans le composant MOM les différents canaux de message (Queue1, Topic1 ou par exemple BonCommande, IHM, …).

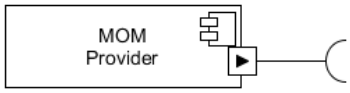
Les producteurs utilisent un port pour chaque canal pour pousser leurs messages.

On reconnaît les consommateurs en mode « Queue » et en mode « Topic » par le type des interfaces de connexion :

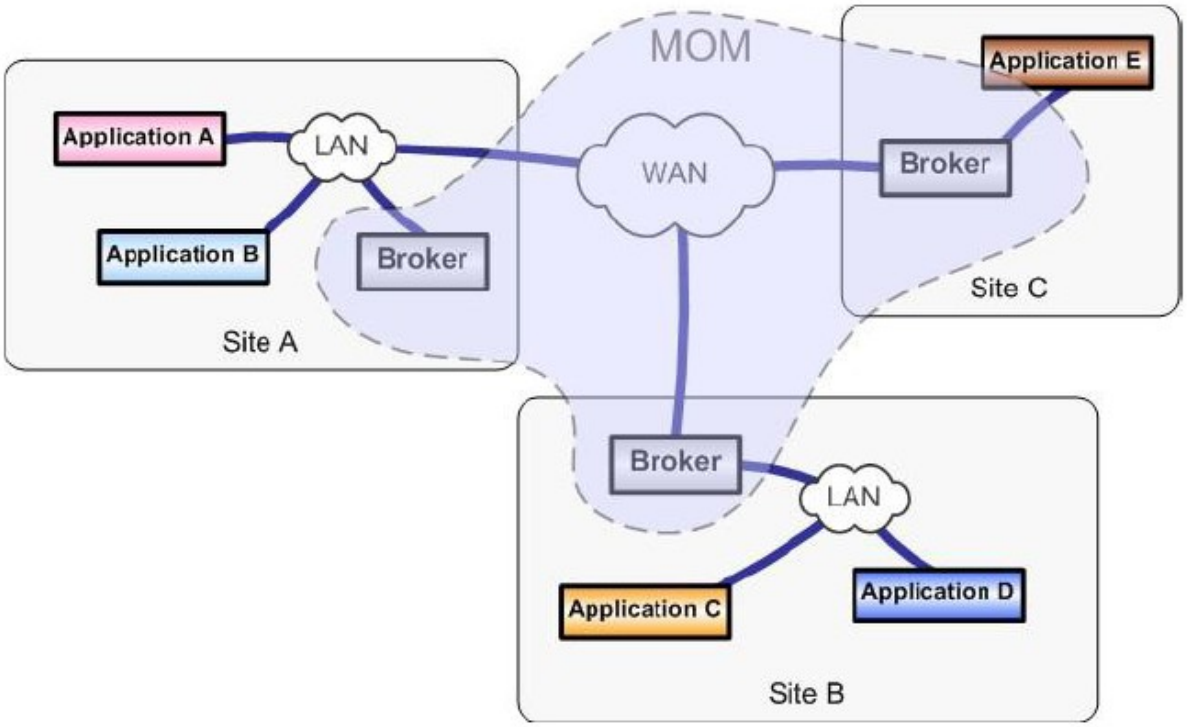
Le mode « Queue » :



Le mode « Topic » :



7.3. Architecture logique des MOM



Les « brokers » servent de passerelles d'échanges entre les différentes applications et constituent donc la structure même d'un MOM.

Ces échanges sont réalisés grâce à un protocole réseau de niveau applicatif et ce protocole définit la représentation des données ainsi que le format des messages et de leurs en-têtes.

Une application cliente s'adresse donc à un « broker » qui va se charger de l'acheminement du messsage sur la bonne file d'attente à destination de ses consommateurs potentiels.

Ce réseau de « brokers » constitue donc le MOM qui est au coeur de l'ensemble des échanges interapplicatifs comme pour tout middleware.

Ainsi il est nécessaire de définir une interface permettant aux applicatifs d'échanger à travers les « brokers » du MOM.

8. Java Messaging System ou JMS

JMS est l'API de JEE permettant aux applications d'utiliser une communication via message dans l'environnement JEE. C'est donc cette API qui permet l'accès à une application d'interfacer avec un MOM.

JMS supporte en particulier le mode point à point gérant des files d'attente et le mode publication abonnement tel que décrit précédemment.

8.1. Terminologie et modèle JMS

Un **JMS Client** (client du Provider) est une application ou un composant Java utilisant les services MOM mis à sa disposition.

Un **JMS Provider** (= intermédiaire) est un service implémentant l'interface JMS et réalisant les mécanismes d'échanges. Il est le composant « intermédiaire » qui orchestre l'échange des messages.

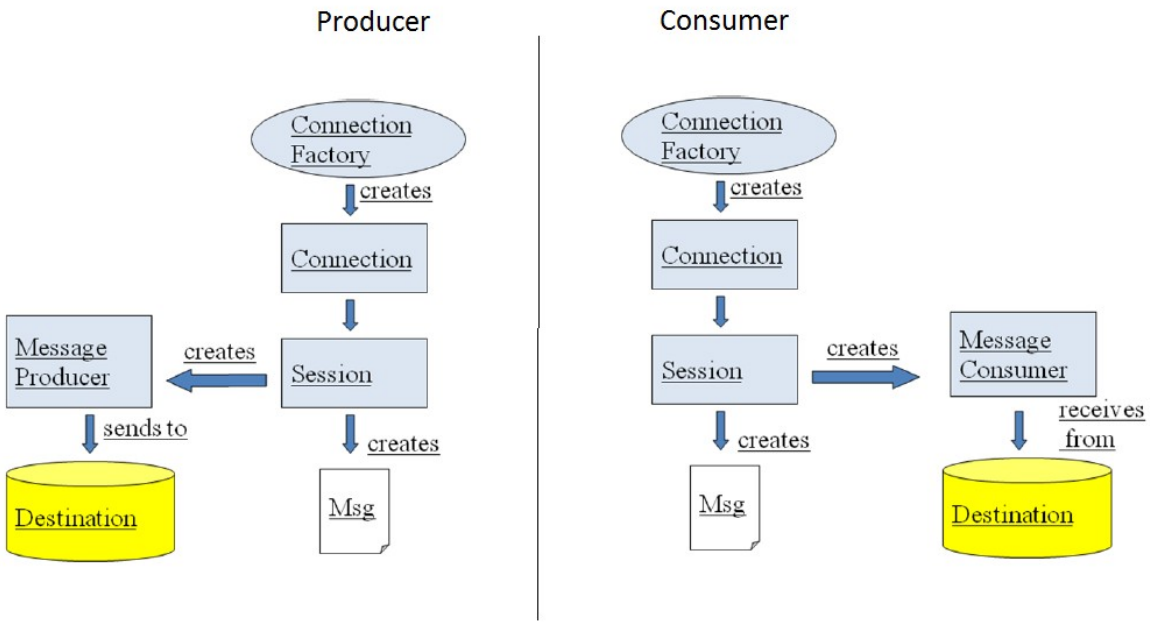
Un **JMS Consumer** est un Client JMS consommant des messages JMS.

Un **JMS Producer** est un Client JMS produisant des messages et les transmettant vers des consommateurs (via l'intermédiaire).
Notons et c'est l'une des particularités intéressantes des MOM, qu'une application peut être à la fois Consumer et Producer ..

Attention à ne pas confondre Client avec Consumer et Provider avec Producer. Consumer et Producer sont tous deux « Clients » d'un Provider.

Un **JMS Message** (= enveloppe) est l'unité d'échange de JMS entre les différents protagonistes

8.2. Modèle de programmation JMS et de l'API associée



Connection : la création d'une connexion permet de relier un Client JMS, un Producer ou un Consumer, au JMS Provider dont l'adresse peut être fournie par l'intermédiaire du service JNDI.

Session : une fois la connexion établie, chaque participant qu'il soit consommateur ou producteur devra créer une session transactionnelle avec le système de gestion de la queue.

Le producteur pourra alors créer des messages et les consommateurs les récupérer par l'intermédiaire des queues ou des topics déjà créés comme le montre le schéma.

Destination : file de message gérée par l'OS

8.3. Création d'un canal

On a la hiérarchie suivante :

- le Provide JMS (java.naming.factory.initial et java.naming.provider.url)
  - le factory de connexion (QueueConnectionFactory) : enregistrée dans l'annuaire JNDI sous le nom *connectionfactoryname*
    - le canal (Queue) enregistrée dans JNDI sous le nom *myQueue ou myTopic* par exemple

Exemple :

Ajout d'une QueueConnectionFactory dans JNDI

```
imqobjmgr add -t qf -l cn=QueueConnectionFactory \
              -j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory \
              -j java.naming.provider.url=file:///var/imq/imq_admin_objects
```

Ajout d'une Queue Queue1 dans JNDI

```
imqobjmgr add -t q -l cn=Queue1 \
              -o imqDestinationName=MyQueue \
              -j java.naming.factory.initial=com.sun.jndi.fscontext.RefFSContextFactory \
              -j java.naming.provider.url=file:///var/imq/imq_admin_objects
```

Ici cn=Queue1 fait référence au nom JNDI de la Queue, à savoir Queue1.
Et imqDestinationName=MyQueue désigne le nom physique de la Queue: MyQueue

8.4. Exemple d'un consommateur « Queue »

Le Producteur :

```
InitialContext jndiContext=new InitialContext();
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);

Connection connection=cf.createConnection();

Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

Destination dest1= jndiContext.lookup("/jms/myQueue");

 […]
MessageProducer producer=session.createProducer(dest1);

Message m=session.createTextMessage();
m.setText("juste un simple message");
producer.send(m);

[…]
connection.close();
```

Le code d'un producteur est le même que le canal soit utilisé en mode « Queue » ou en mode « Topic ».

Le consommateur :

```
InitialContext jndiContext=new InitialContext();
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);

Connection connection=cf.createConnection();

Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

Destination dest1= (Queue) jndiContext.lookup("/jms/myQueue");

MessageConsumer consumer=session.createConsumer(dest1) ;

[…]
Message m=consumer.receive();
```

8.5. Exemple d'un consommateur « Topic »

Le Producteur :

```
InitialContext jndiContext=new InitialContext();
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);

Connection connection=cf.createConnection();

Session session=connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

Destination dest2= jndiContext.lookup("/jms/myTopic");

 […]
MessageProducer producer=session.createProducer(dest2);

Message m=session.createTextMessage();
m.setText("juste un simple message");
producer.send(m);

[…]
connection.close();
```

Remarque : Le code d'un producteur est le même (au nom du canal près) que le canal soit utilisé en mode « Queue » ou en mode « Topic ».

Le consommateur :

```
InitialContext jndiContext=new InitialContext();
ConnectionFactory cf=jndiContext.lookup(connectionfactoryname);

Connection connection=cf.createConnection();

Session                         session=connection.createSession(false,Ses
sion.AUTO_ACKNOWLEDGE);

Destination dest2=(Topic) jndiContext.lookup("/jms/myTopic");
// for publish-subscribe

MessageConsumer consumer=session.createConsumer(dest2) ;

MessageListener listener=new myListener();
consumer.setMessageListener(listener);    // THREAD => asynchrone
```

Dans la classe myListener doit se trouver la methode :

```
public class MysListener implements Listener
{

public void onMessage(Message message) {
  TextMessage msg = null;
  try {
    if (message instanceof TextMessage) {
      msg = (TextMessage) message;
      System.out.println("Reading message: " + msg.getText());
    }
    else {
      System.out.println("Message of wrong type: " +
      message.getClass().getName());
     }
  } catch (JMSException e) {
      System.out.println("JMSException in onMessage(): " + e.toString());
  } catch (Throwable t) {
      System.out.println("Exception in onMessage():" + t.getMessage());
  }
}
```