Programmation d'applications réparties - Application bancaire en Corba

Thomas MIJIEUX - Etienne THIERY

8 janvier 2017

1 Structure globale

1.1 Interface IDL

Comme requis par le sujet, nous avons défini plusieurs structures et interfaces (dans `BankingApp.idl`), reportées ici pour rendre la suite du rapport plus compréhensible.

Une structure de transaction interbancaire, à laquelle on pourrait rajouter des horodatages pour plus de réalisme :

```
struct Transaction
{
    unsigned long dstBankNo;
    unsigned long dstAccountNo;
    unsigned long srcBankNo;
    unsigned long srcAccountNo;
    unsigned long amount;
};

typedef sequence<Transaction> TransactionHistory;
```

Une interface de compte client permettant les opérations élémentaires (consultation, dépot, retrait, transfert) :

```
interface Account
{
    unsigned long getBalance();
    unsigned long getAccountNo();
    unsigned long getBankNo();
    TransactionHistory getHistory();

    void deposit(in unsigned long amount);
    void withdraw(in unsigned long amount);
    void transfer(in unsigned long dstBankNo, in unsigned long dstAccountNo, in unsigned
        long amount);
};
```

Une interface de banque orientée client, permettant de créer un compte et de se connecter. Une interface de banque orientée service interbancaire, définissant un callback appelé lorsqu'un transfert est reçu, et un callback appelé lorsqu'un transfert émis est confirmé. Et une interface générale de banque héritant des deux précédentes (simple solution technique au fait qu'un servant doive étendre une classe et non implémenter une interface, et qu'il ne soit possible d'étendre qu'une seule classe en Java) :

```
interface Bank_CustomerFacing
{
```

```
    unsigned long createAccount(in unsigned long secret);
    Account connect(in unsigned long accountNo, in unsigned long secret);
};

interface Bank_InterbankFacing
{
    boolean initTransaction(in Transaction t);
    void completeTransaction(in Transaction t);
};

interface Bank : Bank_CustomerFacing, Bank_InterbankFacing {};
```

Enfin, une interface de service interbancaire permettant aux banques de s'y connecter, de demander une transaction interbancaire et de récupérer l'historique de ces dernières :

```
interface Interbank
{
    void connect(in unsigned long bankNo, in Bank_InterbankFacing bank);

    void requestTransaction(in unsigned long bankNo, in Transaction t);

    TransactionHistory getHistory(in unsigned long bankNo);
};
```

1.2 Servants

Nous avons développé :
- Un servant implémentant l'interface de service interbancaire (`InterbankServant.java`)
- Un servant implémentant l'interface générale de banque (`BankServant.java`)
- Un servant implémentant l'interface de compte (`AccountServant.java`)

1.3 Serveurs et client

Nous avons également développé :
- Un serveur persistant pour le service interbancaire (`InterbankServer.java`)
- Un serveur éphémère ("transient") pour les banques (`BankServer.java`)
- Un client avec une simple interface en ligne de commande (`Client.java`)

1.4 Compilation et exécution

L'ensemble du projet peut être compilé simplement via la commande :
`make`

L'ensemble des commandes de déploiement peuvent être trouvées dans le fichier `Notice`. Pour tester le projet localement sur une seule machine, il faut lancer au minimum le service interbancaire, 2 banques et 1 client par banque :
- Lancer le serveur interbancaire dans un premier terminal avec `make startInterbank`
- Lancer une banque dans un premier terminal et une autre dans un second terminal avec `make startBank`
- Lancer un client dans un troisième terminal et un autre dans un quatrième avec `make startClient`

Pour tester les différentes opérations il est possible de fermer :
- un serveur bancaire simplement avec `Ctrl-C` dans le terminal correspondant ;
- un client avec `Ctrl-C` ou via une option du menu obtenu une fois connecté ;
- le serveur interbancaire avec `make stopInterbank`

2 Encapsulation

Une des exigences du sujet était que l'interface de compte bancaire ne soit accessible qu'au client et à sa banque, et que le service interbancaire et les autres banques n'y fassent référence que par le numéro de compte.

C'est chose faite : seul le servant de banque instancie un servant de compte à la demande du client et lui retourne une référence sur celui-ci. Les autres banques et le service interbancaire n'y ont pas accès, et font référence au compte via le numéro de banque et le numéro de compte.

3 Intermédiaire bancaire

Une seconde exigence du sujet était la présence d'un intérmédiaire bancaire pour assurer la sincérité des échanges entre deux comptes de banques différentes, et pour tenir un historique des transactions interbancaires.

Nous avons implémenté cela de la façon suivante. Le servant de banque implémente les deux callbacks évoqués plus haut. Lorsque le serveur de banque est lancé, il instancie un servant de banque, puis se connecte au serveur interbancaire, en passant une référence distante sur ce servant, exposant ainsi les deux callbacks.

Puis au moment du transfert :

— (A) Le client demande un transfert via l'interface.
— (B) Le servant du compte émetteur (qui s'exécute sur le serveur de la banque source) appelle une méthode du servant de la banque émettrice (idem), qui demande elle même l'exécution de cette transaction au service interbancaire.
— (C) Le servant du service interbancaire (qui s'exécute lui sur le serveur du service interbanquaire) appelle le callback indiquant la réception d'un transfert à la banque du destinataire.
— (D) Le servant de la banque destinataire (qui s'exécute sur le serveur de la banque destinataire) crédite le compte destinataire, et retourne une validation.
— (E) Le servant du service interbancaire appelle le callback indiquant le succès d'un transfert à la banque de émettrice.
— (F) Le servant de la banque émettrice débite le compte émetteur.
— (G) Le servant du service interbancaire inscrit alors la transaction dans l'historique.

Chaque banque ne peut accéder qu'à la partie de l'historique concernant des transactions de ses clients, et peut donner à un client la sous-partie le concernant à sa demande.

4 Asynchronisme

Une autre exigence du sujet était que les virements s'effectuent de manière asynchrone, c'est à dire qu'ils ne soient pas perdus même si un des 2 serveurs bancaires concernés ne tourne pas.

Les détails de l'implémentation de cet asynchronisme ont été omis dans la partie précédente.

— Si en (B) le serveur émetteur ne tourne pas, le client ne peut pas s'y connecter. On pourrait rendre le serveur bancaire persistant (comme le serveur interbancaire, voir plus bas) de façon à ce qu'il démarre automatiquement, mais nous avons préféré le laisser transient de façon à avoir un exemple de chaque type de serveur dans le projet.
— Si en (B) le serveur interbancaire ne tourne pas, étant implémenté de manière persistante il démarrera automatiquement (voir plus bas).
— Si en (C) le serveur destinataire ne tourne pas, la transaction est insérée dans une queue de transactions en attente sur le serveur interbancaire (qui en a une par serveur bancaire), et la transaction reprendra à cet étape la prochaine fois que le serveur destinataire se connectera à l'interbanque.
— Si en (D) le serveur émetteur ne tourne pas, idem.

5 Persistance

La dernière exigence du sujet était le caractère persistant du service interbancaire, qui devait avoir une référence contante, être instancié automatiquement au démarrage, et avoir une restauration de son état précédent au démarrage.

Nous avons implémenté les deux premiers points avec un POA doté de la politique PERSISTENT, ainsi que les outils `orbd` et `servertool`. On peut tester en stoppant le serveur interbancaire avant d'exécuter un transfert qu'il le relance bien automatiquement.

Nous avons implémenté la restauration automatique de l'état précédent via :
— Une méthode de sauvegarde de l'état dans un fichier (`saveToDisk()`), programmée pour s'exécuter automatiquement à l'arrêt du serveur.
— Une méthode de restauration de l'état par lecture du fichier exécutée à l'instanciation du servant (`restoreFromDisk()`).

Nous avons également implémenté cette restauration pour les serveurs bancaires, sans quoi l'asynchronisme serait inutile : toute transaction en attente serait refusée au relancement du serveur bancaire destinataire ou émetteur puisque ceux-ci auraient "oubliés" les comptes clients correspondants.

6 Limitation

On a supposé que si le serveur interbancaire n'arrive pas à contacter une banque, c'est parce que le serveur de cette dernière ne tourne pas. Ce n'est pas tout à fait réaliste, il pourrait s'agir d'un problème de réseau, et dans ce cas la banque ne se reconnectera pas à l'interbanque une fois la connectivité rétablie. On pourrait résoudre ce problème simplement en mettant en place un démon sur le serveur bancaire qui "ping" le serveur interbancaire en appelant la méthode de connection régulièrement.

Le serveur bancaire n'est pas persistant, donc s'il redémarre alors qu'un client est déjà connecté, le client aura une erreur à sa prochaine requête, puisque la référence du serveur aura changé. Ce n'est pas extrêmement génant puisque le client a juste à se connecter à nouveau, et cela pourrait être résolu avec une politique de POA persistant.

Seules les transactions interbancaires passent par le service interbancaire et sont donc sauvegardées dans son historique. Cela nous semble assez réaliste, mais dans notre implémentation cela implique que l'historique consulté par les clients ne contient que les transactions interbancaires, et pas les transactions intrabancaires. On pourrait implémenter un historique intrabancaire, et faire en sorte que le serveur bancaire fusionne les 2 historiques lorsque le client souhaite consulter ses transactions passées.

On a laissé la possibilité aux clients d'avoir un découvert illimité. Ce n'est pas réaliste, il y a généralement une limite au découvert possible, mais l'implémenter ne nous a pas paru pertinent dans le cadre de ce cours.

On a implémenté une simple connection par mot de passe pour les clients, pas du tout solide d'un point de vue sécurité, mais que nous avons estimé suffisante pour ce cours. Il faudrait quelque chose de bien plus sécurisé pour une application réaliste, et il faudrait également une authentification sécurisée des banques auprès du service interbancaire.

7 Étude de la solution “NotificationService”

On a également essayé de s'intéresser à des solutions s'appuyant sur des fonctionnalités directement disponibles dans la norme CORBA. Notamment, il figure dans la norme CORBA, en plus du NamingService, et du PersistantStateService, deux services nommés EventService et NotificationService (le deuxième étant une extension du premier) qui permettent de délivrer des messages de

manière asynchrone sous forme d'événements (publication d'évènements sur un canal pour l'émetteur et abonnement aux événements pour le récepteur). Après quelques recherches il s'est avéré que la norme du EventService ne définissait aucune notion de persistance [1]. Le NotificationService est une extension du EventService qui vient remédier aux problèmes de l'EventService [2]. En particulier la norme du NotificationService garantie que les événements sont bien délivrés même si un abonné n'est pas disponible lorsque l'événement est publié. On a donc fait une tentative d'implémentation avec le NotificationService (qui n'a malheureusement pas abouti). Les sources peuvent être trouvé dans le dossier `AnnexeNotificationService/`.

Le principe de l'implémentation est le suivant. Les transactions sont vues commes des événements. Le service interbancaire est le "`supplier`" (il publie), et les banques sont les "`consumers`" (elles reçoivent).

Lorsque qu'une banque s'enregistre auprès du service interbancaire, ce dernier crée un `EventChannel`. une référence vers cet `EventChannel` est retournée à la banque via la méthode `set_channel`. Ensuite, le service interbancaire obtient un "`ProxyConsumer`". Le `ProxyConsumer` n'est pas comme on pourrait le croire un consommateur mais en quelque sorte une 'boite d'envoi' pour l'émetteur d'événement. Lorsque que le service interbancaire veut faire parvenir une transaction à une banque, elle encapsule la transaction dans un objet CORBA Any (conteneur universel) et l'envoi grâce à la méthode `push` du `ProxyConsumer`.
Le code relatif à cette partie est disponible dans `AnnexeNotificationService/BankProxy.java` (classe d'objets gérant les banques du point de vue de l'interbanque)

Du point de vue de la banque, une fois enregistrée au près de service interbancaire et la référence sur l'`EventChannel` récupérée, la banque fait deux choses :

- Elle instancie un objet `BankMailBox` qui implémente l'interface `PushSupplier`. Comme son nom l'indique il s'agit d'une boite de reception. Cet objet contient une méthode `push` qui sera appelé par l' `EventChannel` quand un événement surviendra. On pourra dés-encapsuler l'événement pour récuperér la transaction correspondante et intéragir avec la banque de manière adéquate.
- Elle récupère un objet `ProxySupplier` auprès de l'`EventChannel`. Il s'agit d'un objet représentant le `supplier` distant et qui a pour seul rôle de connecter notre boite de reception (`MailBox`) fraîchement instanciée à l'`EventChannel` grâce à une méthode `connect`.

Le code relatif à cette partie est disponible dans `AnnexeNotificationService/BankMailBox.java`.

Différents aspects ont freiné l'aboutissement de cette solution. Le NotificationService est une partie de la norme qui est considéré comme une extension. Ce service n'est pas directement implémenté dans le JDK Java. Un certain nombre d'implémentation de CORBA affirment implémenter le NotificationService mais beaucoup n'implémentent pas tous les aspects et en particulier passent sur les aspects relatifs à la persistance. La seule implémentation affirmant implémenter en Java et de manière complète le `NotificationService` que nous avons su trouver est `OpenORB` [3]. Une des questions qui a été génante pour nous est de savoir s'il est vraiment possible d'utiliser des implémentations CORBA différentes ensembles en Java. Cela semble possible à partir du moment où les interfaces de programmation ont des comportements bien spécifiés, mais comme on l'a vu certaines implémentations n'incluent pas tous les options de la norme. Se reposer complètement sur l'implémentation `OpenORB` semblait aussi extrêmement compliqué. Toutes les recherches effectuées pour tenter de répondre à cet ensemble de questions nous mènent la plupart du temps sur des articles tentant d'expliquer comment bien implémenter certains services CORBA, plutôt que sur des éléments nous permettant réellement de résoudre notre problème initial. En pratique nous n'avons jamais réussi à lancer l'intégralité des programmes de cette solution "`NotificationService`" sans rencontrer de problème bloquant.

1. Implementation of the CORBA Event Service in Java. Paul Stephens. pages 35 à 37(EventService) et pages 38 à 40 (NotificationService) https://www.scss.tcd.ie/publications/tech-reports/reports.99/TCD-CS-1999-34.pdf
2. Section 22.2.3 "Limitations of the EventService" et Section 22.3 "The NotificationService", http://www.ciaranmchale.com/corba-explained-simply/publish-and-subscribe-services.html#sect:event-service:limitations
3. Chapter 1. http://openorb.sourceforge.net/docs/1.3.0/NotificationService/doc/notify.html