5 Janvier 2020

1 Introduction

Dans le cadre de l'UE : *SAR*, dispensée pour les étudiants de M1 MIAGE. Nous avons travaillé en trinôme sur un projet, il s'agit de mettre en place une implantation répartie d'un nouvel algorithme de contrôle de flux. Un site *Platform* est responsable de la gestion du tampon, des sites *Producer*s et *Consumer*s sont responsables de la production et de la consommation de messages.

Ce document accompagne et guide le lecteur à comprendre :

- d'une part, l'algorithme développé (grâce au pseudo-code), la multitude de choix de conception en se référant au diagramme des classes. Ainsi qu'une explicitation et une justification des choix techniques effectués pour arriver à ce résultat.
- et d'autre part, ce document offre *une documentation utilisateur* afin de guider les utilisateurs de l'application et des copies écran d'une simulation.

Le code du projet, ainsi que ce rapport sont disponibles sur le dépôt *Git* à l'adresse : https://github.com/Haouah19/SAR.

2 Rapport technique

2.1 Choix de Conception

La création des classes *Producer*s, *Consumer*s et *Platform*s ne pouvait se faire sans fixer une stratégie de communication des différents éléments de notre architecture. La structure des classes ainsi que l'implémentation des algorithmes dépendent de la manière dont ces dernières communiquent entre elles.

Comme vu durant le second TP réalisé en cours, l'utilisation des sockets permet la communication réseau entre les différentes machines. Les sockets nous permettent de connecter des machines distantes entre elles, d'envoyer et de recevoir des données, d'écouter les communications entrantes. Pour résumer, elles nous permettent de traiter toutes les communications réseaux envisageables entre les machines.

Nous avons commencé par mettre en place un système de communication par sockets mais nous avons très rapidement trouvé des difficultés pour réaliser une

partie importante du projet qui est la circulation des autorisations. L'objectif est de faire circuler des autorisations de production entre les producteurs et le site T et des autorisations de consommation entre les consommateurs et le site T.

L'utilisation des sockets pour communiquer ressemble fortement à une lecture et une écriture dans un fichier, ainsi pour envoyer un message il faut écrire sur le flux de sortie et pour recevoir un message, il faut lire le flux en entrée. Il n'est pas évident en utilisant les sockets d'exécuter une fonction ou des instructions se trouvant dans l'élément récepteur. Le fait que les *Producer*s et les *Consumer*s implémentent *Runnable* rend la tâche encore plus compliquée.

Nous avons essayé d'arriver à un résultat et cela en effectuant des comparaisons sur les messages reçus. Nous avons fixé un nombre de mots-clés qui permettent de savoir ce que *Platform* doit faire. La même stratégie est appliquée dans *Producer* et *Consumer*. Par exemple : un *Producer* voulant produire un message va écrire sur la sortie reliée à *Platform* : **Production**. *Platform* va réceptionner l'élément, et va le comparer aux mots-clés. *Platform* va exécuter le code correspondant, dans ce cas : ça sera la partie qui permet d'attribuer une autorisation à un *Producer*.
Cette manière de faire est complexe et nous avons estimé qu'elle induit une grande complexité temporelle quand il y a un grand nombre de sites en concurrence. et nous n'étions pas tous convaincus de l'efficacité de cette implémentation.

Au fur et à mesure l'utilisation des sockets est devenue compliquée, nous avons jugé qu'il valait mieux que *Producer* et *Consumer* puissent appeler une méthode dans *Platform* quand ils en avaient besoin. Nous avons abandonné l'utilisation des sockets au profit de la technique *RMI*.

L'application développée répond aux exigences de *RMI*. Le cahier des charges nous impose de réaliser une application répartie, pouvant s'exécuter sur plusieurs machines. *RMI* répond à ses problématiques car elle permet l'appel, l'exécution et le renvoi du résultat d'une méthode exécutée dans une machine différente de celle de l'objet l'appelant.

Ainsi l'architecture de l'application a était modifié pour répondre aux exigences de *RMI*, le diagramme des classes de la figure 1 illustre les choix de conception présentés ci-dessous :
Le développement de *Platform* correspond au développement côté serveur de *RMI* :

- *PlatformInterface* hérite de *java.rmi.Remote*, elle contient les méthodes qui peuvent être appelées à distance, vu que la communication entre les *Producer*s et *Consumer*s peut échouer pour diverses raisons, les méthodes peuvent lever l'exception : *java.rmi.RemoteException*.
- *Platform* correspond à l'objet distant, implémente l'interface *PlatformInterface*. Elle met à disposition de *Producer* et *Consumer* les méthodes nécessaires pour arriver à produire et à consommer.

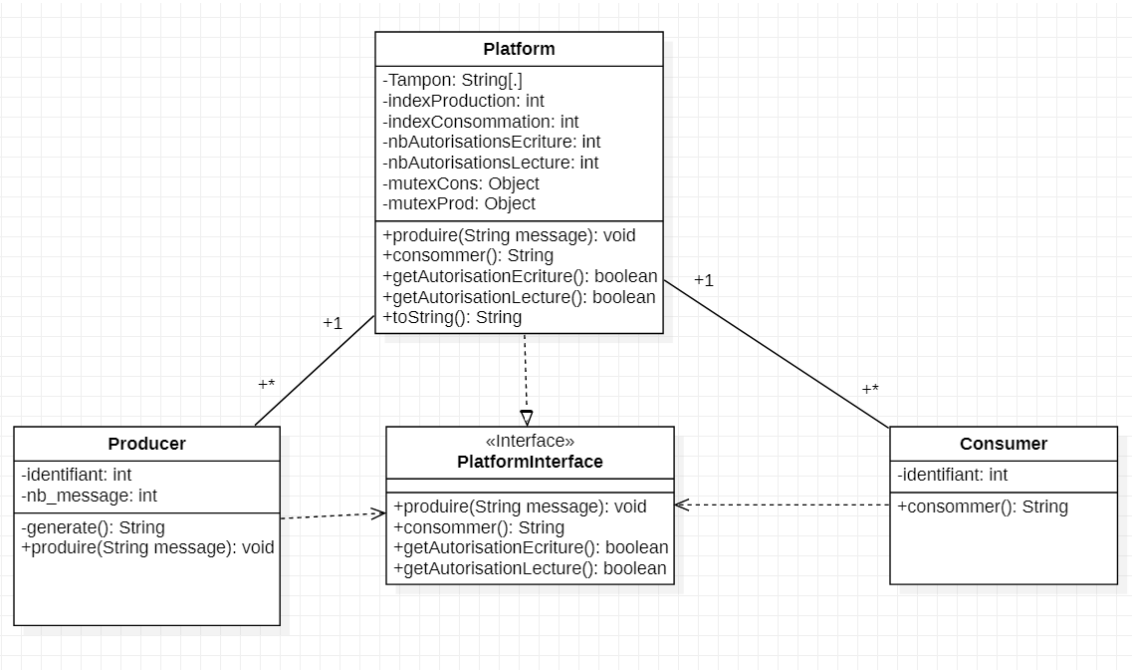

Figure 1: Diagramme de classe simplifié

Platform est aussi responsable de la cohérence du tampon. Il doit être *Thread-Safe*. Un *Producer* ne doit pas écrire si une case est déjà pleine pour ne pas perdre de l'information (*Write-After-Write*) et un *Consumer* ne doit pas s'exécuter en vain, deux *Consumer*s ne doivent pas lire la même valeur. Car il faut savoir que deux clients (*Producer*s ou *Consumer*s) peuvent et exécutent simultanément les méthodes de l'objet distant.

Pour garantir la cohérence du tampon de *Platform*, il nous faut rajouter des mécanismes de synchronisation. Pour cela nous avons créé deux objets *mutexProd* et *mutexCons*, ils protègent les sections critiques. *mutexProd* pour la partie production, et *mutexCons* pour la partie consommation. Ainsi, Il n'y a qu'un seul *Producer*/*Consumer* qui exécute la section critique car il dispose du moniteur de *mutexProd*/*mutexCons*. Le scénario décrit plus haut ne peut arriver, car les sections critiques sont protégées.

- *PlatformServer* permet l'instanciation d'un objet de la classe distante (*Platform*) et son enregistrement dans le registre de noms *RMI*. Cette manœuvre se fait en utilisant la méthode *rebind* de la classe *Naming*.

Le développement de *Producer* et *Consumer* correspond au développement côté client de RMI :

En ce qui concerne le côté client nous avons été obligé de ne pas utiliser de *Sécurity Manager*. La mise en place d'une politique de sécurité posait des problèmes d'accès. En supprimant cet élément, *Producer* et *Consumer* ont accès aux méthodes à distance. Vu le cadre dans lequel l'application a été développé nous avons jugé qu'il est convenable de ne pas utiliser de *Sécurity Manager* côté client.

- *Producer* représente les producteurs. Afin de pouvoir créer facilement un nombre défini de producteurs, *Producer* implémente *Runnable*. Le nombre de *Producer*s est donné par l'utilisateur. Un *Producer* possède :
  - Un *identifiant* unique grâce à un mécanisme de synchronisation présent lors de l'instanciation d'un *Producer*,
  - Une instance de *Platform* qu'il obtient en utilisant la méthode statique *lookup()* de la classe *Naming* présente dans le constructeur de *Producer*,
  - Une méthode *generate()* qui permet de générer des messages.
- *Consumer* représente les consommateurs. Afin de pouvoir créer facilement un nombre défini de consommateurs, *Consumer* implémente *Runnable*. Le nombre de *Consumer*s est donné par l'utilisateur. Un *Consumer* possède :
  - Un *identifiant* unique grâce à un mécanisme de synchronisation présent lors de l'instanciation d'un *Consumer*,
  - Une instance de *Platform* qu'il obtient en utilisant la méthode statique *lookup()* de la classe *Naming* présente dans le constructeur de *Consumer*.

Les *Producer*s et les *Consumer*s tournent infiniment.

2.2 Pseudo Code

- Platform
  Variables :
  - *tampon []* : représente un tableau de *String* de taille N, initialement vide.
  - *indexProduction* : indice d'insertion dans le tampon initialisé à 0
  - *indexConsommation* : indice d'extraction du tampon initialisé à 0
  - *nbAutorisationEcriture* : représente le nombre de *Producer* qui sont autorisé à écrire sur le tampon
  - *nbAutorisationLecture* : représente le nombre de *Consumer* qui sont autorisé à lire dans le tampon

Primitives :

- *produire(String message)* : Elle permet d'insérer l'élement *message* dans le tampon. Cette méthode est appelée par *Producer* quand il désire envoyer un message au *Consumer*.
- *consommer()* : Elle permet d'extraire un élement du le tampon. Cette méthode est appelée par *Consumer* quand il désire lire un message du *Producer*.
- *getAutorisationEcriture()* : Cette methode permet d'attribuer une autorisation d'écriture à un *Producer*. Elle prend en compte les autorisations dèjà émises et le nombre de cases vides dans le tampon pour attribuer ou non une autorisation.
- *getAutorisationLecture()* : Cette methode permet d'attribuer une autorisation de lecture à un *Consumer*. Elle prend en compte les autorisations dèjà émises et le nombre de cases pleines dans le tampon pour attribuer ou non une autorisation.

Pseudo-code :

- *produire(String message)*

```
Début
        tampon[indexProduction] = message;
        IndexProduction = (indexProduction+1)%N;
        nbAutorisationEcriture–;
Fin
```

- *consommer()*

```
Début
        message = tampon[indexConsommation];
        IndexConsommation = (indexConsommation+1)%N;
        nbAutorisationLecture –;
Fin
```

- *getAutorisationEcriture()*

```
Début
        nbCaseVide = 0;
        Pour chaque message du tampon
                Si (message == null)
                        nbCaseVide++;
                Fin si
        Fin pour
        Si (nbCaseVide strict supérieur à nbAutorisationEcriture)
                nbAutorisationEcriture++;
                Retourner True;
        Fin si
        Retourner False ;
Fin
```

- *getAutorisationLecture()*

```
Début
```

nbCasePleine = 0;
Pour chaque message du tampon
Si (message != null)
nbCasePleine++;
Fin si
Fin pour
Si (nbCasePleine strict supérieur à nbAutorisationLecture)
nbAutorisationLecture++;
Retourner *True*;
Fin si
Retourner *False* ;
Fin

- Producer
  Variables :
  - *identifiant* : Variable unique d'identification d'un *Producer*, son instantiation est *Thread-Safe*.
  - *tampon* : Référence vers *Platform*, C'est un objet distant récuperé grâce à l'architecture RMI.
  - *nb_message* : Nombre de message produit, initialisé à 0.

  Primitives :
  - *generate()* : permet de génerer des messages afin de les envoyer aux *Consumer*s. Ces messages ont un format préci *identifiant* + *nb_message*. Cela nous permet de garder une traçabilité.
  - *produire(String message)* : Cette methode fait appel à la méthode *produire()* de *Platform* et cela grâce à la variable *tampon*.
  - *run()* Vu que *Producer* implemente *Runnable*, la classe doit redéfinir la methode *run()*. Elle contient les traitements à exécuter.

  Pseudo-code :
  - *generate()*
    Début
    nb_message ++;
    message = identifiant, "_", nb_message;
    retourner message;
    Fin
  - *produire(String message)*
    Début
    tampon.produire(message);
    Fin
  - *run()*
    Début
    Tant que (*True*)
    Si (tampon.getAutorisationEcriture() == *True*)
    produire(generate());

Fin si
Fin Tant que
Fin

- Consumer
  Variables :
  - *identifiant* : Variable unique d'identification d'un *Consumer*son instantiation est *Thread-Safe*.
  - *tampon* : Référence vers *Platform*, C'est un objet distant récuperé grâce à l'architecture RMI.

  Primitives :
  - *consommer()* : Cette methode fait appel à la méthode *Consommer()* de *Platform* et cela grâce à la variable *tampon*.
  - *run()* : Vu que *Consumer* implemente *Runnable*, la classe doit redéfinir la methode *run()*. Elle contient les traitements à exécuter.

  Pseudo-code :
  - *consommer()*
    Début
    message = tampon.consommer();
    retourner message;
    Fin
  - *run()*
    Début
    Tant que (*True*)
    Si (tampon.getAutorisationLecture() == *True*)
    message = consommer();
    afficher message
    Fin si
    Fin Tant que
    Fin

3 Documentation utilisateur

1. Récupéreration du code source de l'application :
   - en utilisant Git :
     - Ouvrez un terminal
     - Tappez la commande suivante : git clone https://github.com/Haouah19/SAR.git
   - en décompressant une archive ZIP.
2. Installation et lancement de l'application :
   - En local :
     - Ouvrez trois terminaux

- Sur l'un des terminals, allez dans le dossier src/platform
  (a) Compilez les sources avec : javac PlatformInterface.java Platform.java PlatformServer.java
  (b) Créez les talons pour les objets appelés à distance : rmic -vcompat Platform
  (c) Pour activer le service de noms, il faut spécifier le fichier de sécurité comme suit : rmiregistry -J-Djava.security.policy=serveur.policy 2001 &
  (d) lancez la *Platform* : java PlatformServer

- Sur le second terminal, allez dans le dossier src/producer
  (a) Tappez la commande suivante : javac PlatformInterface.java Producer.java
  (b) lancer les *Producer*s : java Producer

- Sur le dernier terminal, allez dans le dossier src/consumer
  (a) Tappez la commande suivante : javac PlatformInterface.java Consumer.java
  (b) lancer les *Consumer*s : java Consumer

• À distance :

- Choisissez une machine qui va devenir le serveur RMI.
  Remarque : cette machine doit rester allumer pour que les *Producer*s et *Consumer*s puissent s'exécuter. Si le code de *Platform* arrête de s'exécuter, les *Producer*s et *Consumer*s stopperont leurs exécutions en signalant : *Problème lors de l'exécution, une RemoteException a été levé !*
  (a) allez dans le dossier src/platform
  (b) Compilez les sources avec : javac PlatformInterface.java Platform.java PlatformServer.java
  (c) Créez les talons pour les objets appelés à distance : rmic -vcompat Platform
  (d) Pour activer le service de noms, il faut spécifier le fichier de sécurité comme suit : rmiregistry -J-Djava.security.policy=serveur.policy 2001 &
  (e) lancez la *Platform* : java PlatformServer (*PlatformServer* va mettre dans son registre l'addresse IP de la machine)

- Sur le deuxième machine, allez dans le dossier src/producer
  (a) Tappez la commande suivante : javac PlatformInterface.java Producer.java
  (b) lancer les *Producer*s : java Producer @Ip de la machine 1

– Sur le dernier terminal, allez dans le dossier src/consumer
  (a) Tappez la commande suivante : javac PlatformInterface.java Consumer.java
  (b) lancer les *Consumer*s : java Consumer @Ip de la machine 1

3. Exécution de l'application

   • Nous avons décidé de ne pas détailler cette partie, car arrivé à cette étape, l'application explique au fur et à mesure les informations nécessaires pour l'exécution.

   Si vous rencontrez un problème pour reproduire les étapes décrites ci-dessus veuillez envoyer un mail à l'adresse : ahmed_haouili@yahoo.fr

4 Quelques exécutions

1. Lancement de *Platform* : **voir Figure 2**
   L'utilisateur choisit la taille du tampom, ici on prend une taille de 4.

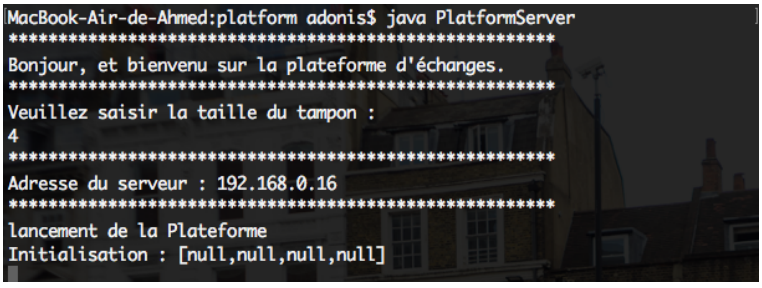

Figure 2: Lancement de Platform

2. Lancement de *Producer* : **voir Figure 3**

3. lancement de *Consumer* : L'utilisateur choisit le nombre de consommateurs, ici on crée 3 consommateurs. **voir Figure 4**

4. **Figure 5** : les *Producer*s et les *Consumer*s écrivent et lisent d'une manière circulaire. le tampon est vide au début de l'exécution et à la fin de la production.
   Figure 6 : S'il n'y a pas eu d'interruption de réseau, à la fin de l'exécution des *Producer*s le tampon sera vide et les *Consumer*s auront lu tous les messages.

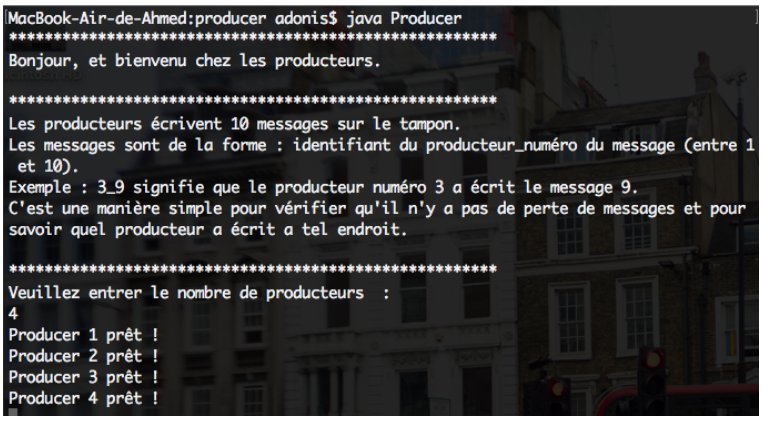

Figure 3: Lancement de Producer

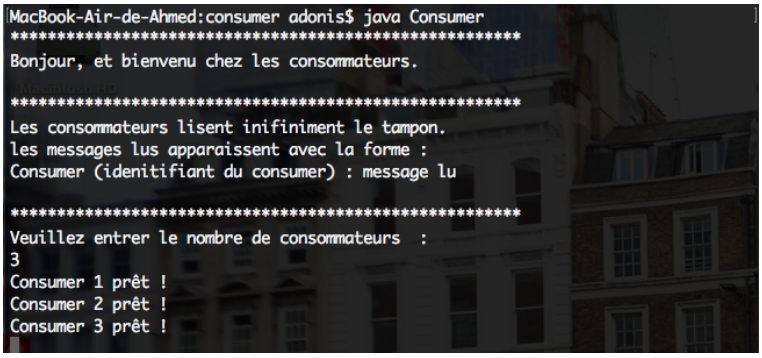

Figure 4: Lancement de Consumer

Figure 5: Algo en cours d'exécution

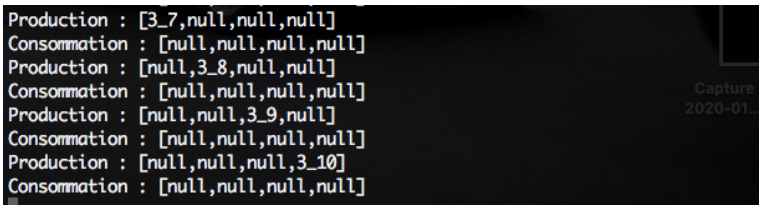

Figure 6: Algo fini