M4102 : Programmation Répartie
Travaux Pratiques 1
Développement d'une invocation de méthode distante

Ce TP, va nous permettre d'illustrer les bases des systèmes d'invocation d'objets répartis. Pour cela, nous allons partir d'un service simple : une classe qui fournit des méthodes sur des chaînes de caractères et d'une classe utilisatrice. Nous allons introduire progressivement des classes intermédiaires jusqu'à pouvoir rendre notre service accessible à distance.

1. **Pour commencer**

   Nous allons partir d'un code déjà écrit qui comprend :
   - Une interface `AlaChaineInterface` qui spécifie les méthodes du *service* proposé.
   - Une classe `AlaChaine` qui implémente le *service* qu'on veut rendre accessible à distance.
   - Une classe `Utilisatrice` qui servira de *client*.
   - Une classe `Intermediaire` qui introduit un premier découplage entre le *client* et le *service*. **Cette classe doit avoir la même interface que le service** pour garder un code d'utilisation identique dans la classe `Utilisatrice`.

   La figure 1 illustre la modification de l'invocation avec une classe intermédiaire. Hormis l'instanciation, l'appel des méthodes reste strictement identique pour la classe `Utilisatrice`.

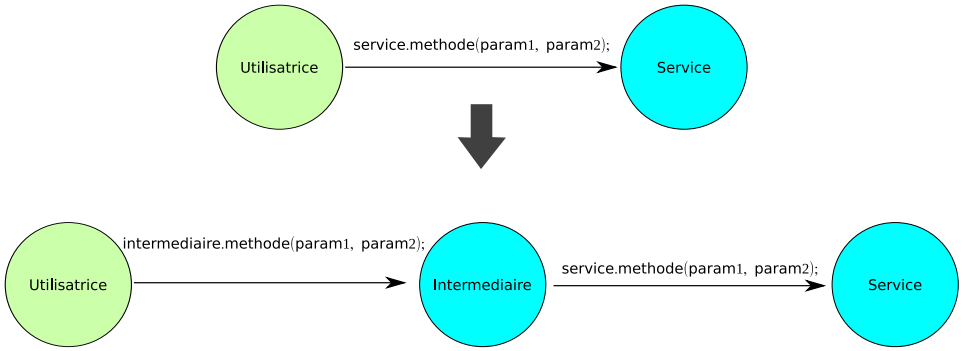

FIGURE 1 – introduction d'un intermédiaire dans l'invocation

   (a) Placer vous dans un répertoire de développement et clonez le point de départ du TP avec la commande suivante [1] :
   `git clone https://git-iut.univ-lille.fr/peter/m4102_tp1.git`

   (b) Tester le code en exécutant la classe `Utilisatrice` puis modifiez la classe `Intermédiaire` afin qu'elle affiche pour chaque appel :
   - la méthode appelée
   - le type et la valeur de chaque paramètre
   - le type et la valeur du résultat
   - le type et le message s'il y a eu une exception

   (c) Faite un `commit` de code.

1. Vous aller récupérer des classes qui nous serviront pour la suite du TP. Vous pouvez les ignorer pour l'instant.

2. **Définition d'un protocole de communication**

   Notre objectif à terme est de permettre que le *client* et le *service* puissent être sur des machines différentes. Pour préparer cela, il faut une classe intermédiaire appelée par le client et une classe côté serveur pour appeler ses méthodes. Ces deux classes serviront à cacher cette répartition sur des machines différentes (voir figure 2).

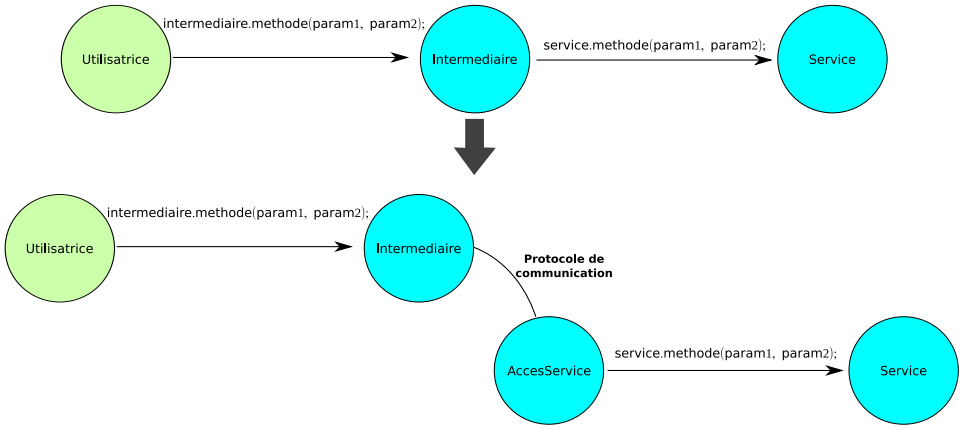

FIGURE 2 – Préparation de l'invocation à distance

   La transmission des invocations entre la classe `Intermediaire` et la classe `AccesService` se fera à terme avec une socket TCP. On va définir un protocole orienté texte pour représenter les invocations de méthodes et leurs résultats. Ce protocole va définir quels sont les messages qui peuvent être échangés entre les deux classes et comment ils seront écrits (format des messages).

   (a) Vous trouverez dans le fichier `Readme.md` une proposition de protocole pour l'invocation d'une méthode distante utilisant la forme de Backus-Naur (BNF). Complétez le fichier `Readme.md` avec une définition du format de message nécessaire pour transmettre la réponse.

   (b) Vous allez maintenant implementer votre protocole. Pour cela :
   - modifiez votre classe `Intermediaire` pour quelle transmette à la classe `AccesService` la représentation de l'invocation de méthode sous forme de chaîne de caractères et qu'elle interprète la chaîne reçue en retour pour renvoyer le résultat au client ;
   - Complétez la classe `AccesService` pour qu'elle réalise l'invocation des méthodes de la classe `AlaChaine` à partir de la chaîne reçue et retourne le résultat.

   (c) Quand c'est terminé, faite un `commit` de votre code.

3. **Répartition du client et du serveur**

   On a maintenant tous les éléments pour réaliser notre répartition. Il ne nous manque plus qu'un client et un serveur socket (figure 3).

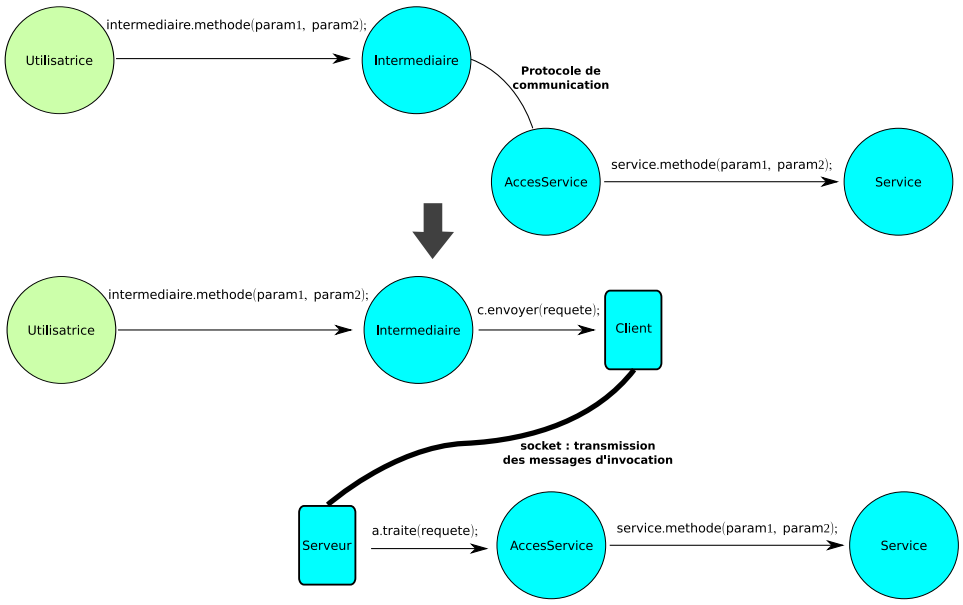

FIGURE 3 – Invocation distante à travers une socket

(a) Dans le projet, vous trouverez une classe `Serveur` qui fournit le code de socket côté serveur. Cette classe devra instancier la classe `AccesService` et lui passer en paramètre la socket du client.

(b) Modifiez la classe `AccesService` pour quelle utilise cette socket pour récupérer l'invocation et transmettre le résultat via la socket.

(c) Pour tester le fonctionnement de votre code, vous pouvez utiliser la commande `nc` [2] qui vous permettra de vous connecter et d'envoyer «à la main» vos requêtes au serveur.

(d) Dans le projet, vous trouverez une classe `Client` qui fournit le code socket côté client. Utilisez cette classe dans la classe `Intermediaire` pour envoyer l'information dans la socket et récupérer le résultat.

(e) Quand c'est terminé, faite un `commit` de votre code.

4. **Servir plusieurs clients**

   Pour l'instant notre serveur ne peut répondre qu'à une invocation à la fois. Modifier votre code en utilisant des `Thread` pour pouvoir répondre à plusieurs clients. Pour cela, il faudra :

   — Modifier la classe `AccesService` pour qu'elle implémente `Runnable` ou hérite de `Thread` et implémenter la méthode `run()`.

   — Modifier la classe `Serveur` afin qu'elle instancie un nouveau `Thread` à chaque connexion.

2. Pour envoyer votre requête au serveur utilisez CTRL + D plutôt que Entrée qui envoie un retour chariot.

Quand c'est terminé, faite un `commit` de votre code.

Rendu et évaluation du TP

Ce TP sera rendu sous forme d'un projet sur le gitlab de l'IUT. Pour cela, à la fin du TP, vous devrez réaliser les opérations suivantes :

1. Allez sur le gitlab de l'IUT et créez un projet nommé `M4102_tp1`
2. sur ce projet gitlab, ajoutez votre enseignant de TP et l'enseignant de cours avec le statut développeur pour qu'ils puissent accéder à votre code.
3. Placez vous dans le répertoire où vous avez codé et tapez les commandes suivantes :
   — `git remote rename origin old-origin`
   — `git remote add origin  https://git-iut.univ-lille.fr/login/M4102_tp1.git` où `login` correspond à votre nom de login.
   — `git push -u origin --all`

Ce TP sera évalué sur la définition du protocole (fichier `Readme.md`) ainsi que sur votre implémentation. Le barême proposé est le suivant :

— protocole : 2 points
— implémentation correcte du protocole d'invocation : 1 point
— gestion correcte des réponses aux invocations : 1 point
— communication de l'invocation par Socket : 1 point
— gestion de plusieurs clients simultanés : 1 point
