

Compte-Rendu TP6

Programmation Répartie - ONC RPC

REDJRADJ YACINE GROUPE A

12 avril 2026

Table des matières

1	Rappel du sujet	2
2	Analyse du sujet	2
2.1	Exercice 1 : Serveur de Calcul	2
2.2	Exercice 2 : Serveur de Chat	2
3	Choix techniques effectués	2
4	Résultats et tests	3
4.1	Serveur de calcul	3
4.2	Serveur de chat	3
5	Conclusion	4

1 Rappel du sujet

L'objectif de ce TP est de se familiariser avec la mise en œuvre de systèmes distribués en utilisant le protocole ONC RPC en environnement Java, via la librairie Remote Tea.

Le TP s'articule autour de deux exercices principaux :

1. **Un serveur de calcul** : Implémentation d'un service RPC capable d'effectuer des opérations arithmétiques basiques (addition, soustraction, multiplication, division, modulo) sur des entiers et des flottants, en gérant plusieurs versions d'un même programme RPC.
2. **Un serveur de chat** : Conception et développement d'un système de discussion en temps réel. Le système repose sur une architecture où le serveur central diffuse les messages à l'ensemble des clients connectés. Les clients agissent également comme des serveurs secondaires pour recevoir ces notifications asynchrones.

2 Analyse du sujet

2.1 Exercice 1 : Serveur de Calcul

Le problème principal réside dans la définition de l'interface en langage RPCL (.x) et la gestion des deux versions (Int et Float) du service. Une fois le squelette généré par `jrpcgen`, l'enjeu est de surcharger les méthodes distantes côté serveur pour implémenter la logique métier, et d'invoquer ces méthodes côté client de manière transparente.

2.2 Exercice 2 : Serveur de Chat

Contrairement à une requête classique client-serveur (synchrone), un chat requiert que le serveur puisse contacter le client de manière asynchrone dès qu'un nouveau message est reçu. L'architecture nécessite donc une communication bidirectionnelle :

- Le **serveur principal** qui écoute sur un port fixe et maintient une liste des clients connectés (adresse IP et port d'écoute).
- Le **client principal** qui initie une connexion, envoie ses messages au serveur central, mais déploie également un **serveur secondaire** (dans un thread séparé) écoutant sur un port dynamique pour recevoir en continu les messages diffusés.

3 Choix techniques effectués

Pour répondre aux contraintes du sujet, plusieurs choix techniques ont été réalisés :

- **Framework et librairies** : Utilisation de la librairie Java `Remote Tea` pour l'implémentation du protocole ONC RPC. La compilation utilise `jrpcgen.jar` pour générer les stubs et les skeletons, et `oncrpc.jar` pour gérer l'exécution.
- **Serveur de calcul** : Le fichier .x définit un seul numéro de programme (0x22222220) et deux versions (1 pour entiers, 2 pour flottants). Le serveur hérite du skeleton généré et implémente chaque fonction arithmétique.
- **Architecture du Chat** :
 - Le serveur central stocke les informations de chaque client sous la forme d'une chaîne de caractères "AdresseIP:Port".

- Lors du démarrage d'un client, celui-ci choisit aléatoirement un port d'écoute unique et lance un thread serveur local pour récupérer les messages.
- Lors d'un appel à `ENVOI_1` par un client, le serveur principal agit en tant que client RPC pour contacter successivement chaque adresse enregistrée et appeler leur méthode `RECEPTION_1`.
- Utilisation du protocole TCP (`OncRpcProtocols.ONCRPC_TCP`) pour assurer la fiabilité des échanges et le bon ordre des messages de chat.

4 Résultats et tests

Des batteries de tests ont été effectuées pour s'assurer du bon fonctionnement des deux programmes :

4.1 Serveur de calcul

Le client est capable de se connecter au serveur (préalablement lancé et enregistré auprès de `jportmap`) et d'invoquer avec succès les calculs sur les deux versions (entiers et flottants). Les résultats retournés par le serveur sont corrects, démontrant ainsi la bonne sérialisation et transfert des données (XDR).

```

Last login: Mon Apr 13 08:28:56 on tty005
yacinredjrad@MacBook-Air-de-Yacine CalculRPC % java -classpath ./oncrpc.jar:./jportmap.jar:./jportmap.jar:./CalculClientMain
Client de calcul connecte au serveur.
Choisissez le type d'operation :
1 - Addition int
2 - Soustraction int
3 - Multiplication int
4 - Division int
5 - Modulo int
6 - Addition float
7 - Soustraction float
8 - Multiplication float
9 - Division float
0 - Quitter
> 1
a = 5
b = 4
Resultat : 9
Choisissez le type d'operation :
1 - Addition int
2 - Soustraction int
3 - Multiplication int
4 - Division int
5 - Modulo int
6 - Addition float
7 - Soustraction float
8 - Multiplication float
9 - Division float
0 - Quitter
> 6
a = 5
b = 2
Resultat : 2.5
Choisissez le type d'operation :
1 - Addition int
2 - Soustraction int
3 - Multiplication int
4 - Division int
5 - Modulo int
6 - Addition float
7 - Soustraction float
8 - Multiplication float
9 - Division float
0 - Quitter
>

```

FIGURE 1 – Test du client de calcul avec le menu interactif

4.2 Serveur de chat

Le scénario de test suivant a été validé :

1. Lancement de `jportmap` en tâche de fond.
2. Lancement du serveur de chat `ChatServerMain`.
3. Lancement d'un premier client, saisie de l'IP (`localhost`), génération d'un port dynamique, et enregistrement au serveur avec le pseudo "Alice".
4. Lancement d'un deuxième client sur le même principe avec le pseudo "Bob".

5. L'envoi d'un message par "Alice" est immédiatement reçu et notifié au serveur principal qui le redistribue, via un appel RPC inversé, au client "Bob" et au client "Alice".

Le système se comporte de manière stable. Plusieurs terminaux locaux ont été utilisés pour simuler un contexte fortement concurrent.

```

Terminal Shell Édition Présentation Fenêtre Aide
ChatRPC - java -jar jportmap.jar - 102x27
Last login: Mon Apr 13 00:34:02 on ttys006
yacineredjradj@MacBook-Air-de-Yacine ChatRPC % java -jar jportmap.jar
[]

ChatRPC - java -classpath ./oncrpc.jar:./jrpcgen.jar:./jportmap.jar: ChatClientMain - 102x27
Last login: Mon Apr 13 00:34:06 on ttys007
yacineredjradj@MacBook-Air-de-Yacine ChatRPC % java -classpath ./oncrpc.jar:./jrpcgen.jar:./jportmap.jar: ChatClientMain
Entrez votre adresse IP (ou localhost si test local) : localhost
Votre pseudo : Alice
Salut Bob
[Message] Alice : Salut Bob
[Message] Bob : Salut Alice
[Message] Bob : ça va ?
oui et toi ?
[Message] Alice : oui et toi ?

ChatRPC - java -classpath ./oncrpc.jar:./jrpcgen.jar: ChatServerMain - 102x27
Last login: Mon Apr 13 00:33:57 on ttys006
yacineredjradj@MacBook-Air-de-Yacine ChatRPC % java -classpath ./oncrpc.jar:./jrpcgen.jar:./jportmap.jar: ChatServerMain
Serveur de chat démarré et en attente de connexions...
Nouveau client connecté : localhost:4563
Nouveau client connecté : localhost:4251
Message reçu : Alice : Salut Bob
Message reçu : Bob : Salut Alice
Message reçu : Bob : ça va ?
Message reçu : Alice : oui et toi ?

ChatRPC - java -classpath ./oncrpc.jar:./jrpcgen.jar:./jportmap.jar: ChatClientMain - 102x27
Last login: Mon Apr 13 00:29:47 on ttys007
yacineredjradj@MacBook-Air-de-Yacine ChatRPC % java -classpath ./oncrpc.jar:./jrpcgen.jar:./jportmap.jar: ChatClientMain
Entrez votre adresse IP (ou localhost si test local) : localhost
Votre pseudo : Bob
[Message] Alice : Salut Bob
Salut Alice
[Message] Bob : Salut Alice
ça va ?
[Message] Bob : ça va ?
[Message] Alice : oui et toi ?

```

FIGURE 2 – Test de communication bidirectionnelle entre le serveur et les clients

5 Conclusion

Ce TP a permis d'assimiler les principes fondamentaux de la programmation répartie avec ONC RPC en Java, ainsi que l'utilisation indispensable d'outils de sérialisation pour le passage d'arguments et du *portmapper* pour l'allocation dynamique des ports.

Difficultés rencontrées :

La mise en place de l'architecture bidirectionnelle pour le serveur de chat a représenté la principale difficulté. En effet, il a fallu concevoir un "serveur dans le client" tournant en arrière-plan pour recevoir les *callbacks* du serveur central, le tout en gérant correctement les *threads* pour ne pas bloquer l'écoute de la saisie utilisateur au clavier.

Limites et perspectives d'évolution :

- **Déconnexion des clients :** Le système de chat actuel ne prend pas en compte la déconnexion volontaire ou brutale d'un client. Si un client quitte le chat, le serveur essaiera de lui envoyer des messages, ce qui génère une exception et sature le réseau. Une fonction RPC supplémentaire `DECONNEXION_1` pourrait régler ce problème.
- **Richness des messages :** Le passage de paramètres complexes (objet message contenant la date, l'auteur, etc., définis via des `struct` en RPCL) au lieu d'une simple chaîne de caractères concaténée permettrait de structurer et d'enrichir l'application.

- **Interface Graphique :** L'ajout d'une IHM (via Swing ou JavaFX) pour le client de chat rendrait le système beaucoup plus convivial pour les utilisateurs finaux.