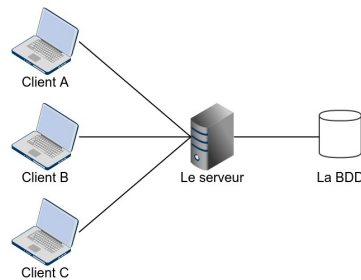


Rapport de Projet - Système de Gestion de Base de Donnée Orientée Objet



Réalisé par :

Ryan LAWSON

Dans le cadre du cours d'Info 4B

Département IEM

Université de Bourgogne

Avril 2025

Table des matières

Introduction	3
1 Analyse Fonctionnelle	4
1.1 Règles de fonctionnement	4
1.1.1 Commandes client	4
1.1.2 Gestion des sessions	4
1.1.3 Persistance des données	4
1.2 Découpage en sous-problèmes	4
1.2.1 Communication réseau	4
1.2.2 Gestion des données	5
1.2.3 Authentification	5
1.3 Diagramme d'architecture	5
2 Structures de Données	6
2.1 Choix des structures	6
2.1.1 ConcurrentHashMap et List<ObjetBDD>	6
2.1.2 Justification des collections Java	6
2.2 Spécification des classes	7
2.2.1 ClientDB et Serveur	7
2.2.2 Message et Collection	7
3 Architecture en Couches	8
3.1 Couche réseau	8
3.1.1 Gestion des sockets	8
3.1.2 Sérialisation des messages	8
3.2 Couche données	8
3.2.1 Stockage des objets	8
3.2.2 Recherche et filtrage	9
3.3 Couche sécurité	9
3.3.1 GestionnaireAuth	9
3.3.2 Hachage des mots de passe	9
4 Algorithmes Principaux	10
4.1 Recherche d'objets	10
4.1.1 Algorithme de filtrage	10
4.1.2 Optimisation des performances	10
4.2 Authentification	11
4.2.1 Génération de sessionId	11
4.2.2 Validation des identifiants	11

5	Jeu de Tests	12
5.1	Scénarios de test	12
5.1.1	Test de connexion	12
5.1.2	Test de persistance	12
5.2	Résultats	12
5.2.1	Sorties client/serveur	12
5.2.2	Validation des fonctionnalités	12
	Conclusion	16

Introduction

Contexte et objectifs

Ce projet, réalisé dans le cadre du cours d'Info 4B (Principes des systèmes d'exploitation), a pour but de concevoir un système client-serveur de gestion de collections d'objets sérialisés. Développé intégralement en solo, il met en œuvre une architecture modulaire inspirée des principes des systèmes d'exploitation, avec les objectifs suivants :

- Fournir une interface client en ligne de commande pour interagir avec un serveur
- Garantir la persistance des données via la sérialisation Java
- Implémenter un mécanisme d'authentification sécurisé
- Respecter une conception en couches fonctionnelles (réseau, données, sécurité)

Fonctionnalités clés

Le système propose les fonctionnalités principales suivantes :

- **Gestion des collections** : Création, suppression et listage
- **Manipulation d'objets** : Ajout, recherche et suppression d'objets `ObjetBDD`
- **Sécurité** : Authentification utilisateur avec gestion de sessions via UUID
- **Persistance** : Sauvegarde automatique dans des fichiers `.ser`

Approche technique

L'architecture repose sur :

- Une **couche réseau** utilisant des sockets Java et des messages sérialisés (`Message`)
- Une **couche données** avec des collections génériques (`Collection<T>`)
- Une **couche sécurité** via le `GestionnaireAuth` et le hachage de mots de passe
- Des structures de données thread-safe (`ConcurrentHashMap`, `ArrayList`)

Structure du rapport

Ce document détaille :

1. L'analyse fonctionnelle et le découpage modulaire
2. Les choix techniques justifiés (structures de données, algorithmes)
3. L'architecture en couches alignée avec les concepts des systèmes d'exploitation
4. Les tests validant les scénarios d'usage critiques

Chapitre 1

Analyse Fonctionnelle

1.1 Règles de fonctionnement

1.1.1 Commandes client

Le système propose un ensemble de commandes en ligne de commande pour interagir avec le serveur :

- **CONNECT** <hôte> <port> : Établit une connexion au serveur
- **CREATE_COLLECTION** <nom> : Crée une nouvelle collection
- **LOGIN** <user> <password> : Authentifie l'utilisateur et crée une session
- **SEARCH** <collection> <critère> : Recherche des objets par critère
- **EXIT** : Ferme proprement la connexion

Contraintes techniques :

- Toute commande (sauf **CONNECT**) nécessite une connexion active
- Les requêtes non authentifiées (**LOGIN**) sont rejetées
- Format des messages : Objets Java sérialisés (**Message**)

1.1.2 Gestion des sessions

- Session ID généré via `UUID.randomUUID().toString()`
- Durée de vie : Valide jusqu'au **LOGOUT** explicite ou déconnexion
- Stockage : `ConcurrentHashMap` dans `GestionnaireAuth` (thread-safe)

1.1.3 Persistance des données

- Format de stockage : Fichiers `.ser` (sérialisation Java)
- Automatisation : Sauvegarde après chaque modification (`GestionnaireCollections`)
- Chargement au démarrage du serveur

1.2 Découpage en sous-problèmes

1.2.1 Communication réseau

- **Classes clés** : `ClientDB`, `Serveur`, `GestionnaireClient`
- **Fonctionnalités** :
 - Gestion des sockets TCP/IP
 - Sérialisation/désérialisation des objets **Message**
 - Gestion multi-thread des clients (**Thread**)

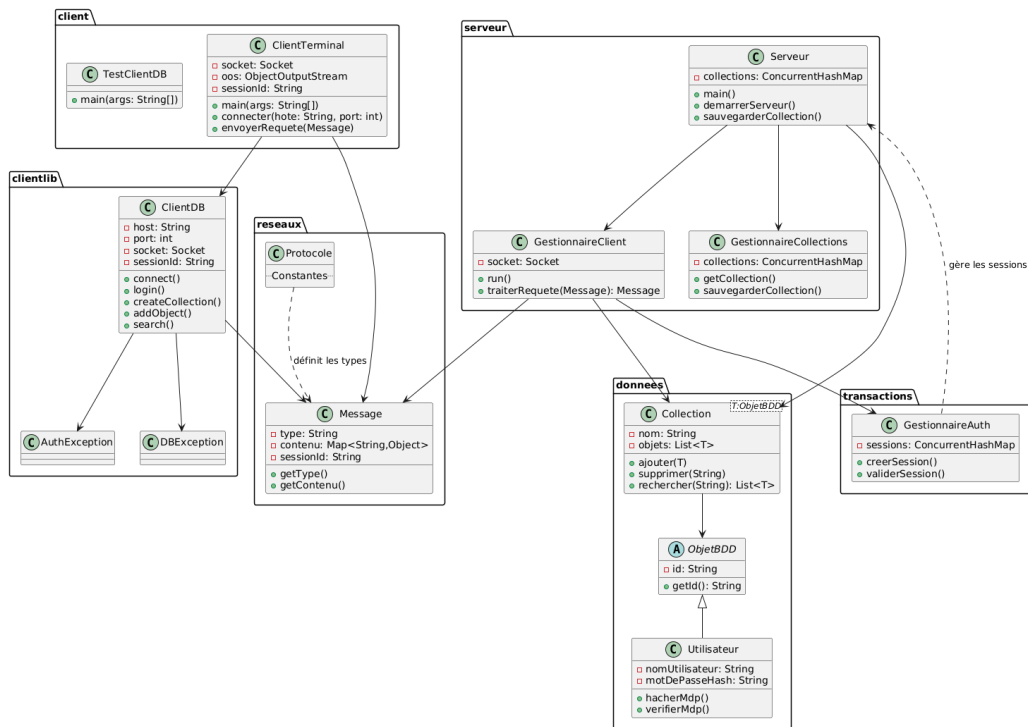
1.2.2 Gestion des données

- **Classes clés** : `Collection<T>`, `ObjetBDD`
- **Fonctionnalités** :
 - CRUD sur les collections
 - Recherche par critères (`id` ou `nomUtilisateur`)
 - Généricité via le type `T` extends `ObjetBDD`

1.2.3 Authentification

- **Classes clés** : `GestionnaireAuth`, `Utilisateur`
- **Mécanismes** :
 - Hachage MD5 simplifié (via `String.hashCode()`)
 - Table de sessions concurrente (`ConcurrentHashMap`)
 - Validation systématique du `sessionId` avant chaque opération

1.3 Diagramme d'architecture



Légende :

- Flèches pleines : Dépendances directes
- Flèches pointillées : Flux de données
- Rectangles : Couches fonctionnelles

Chapitre 2

Structures de Données

2.1 Choix des structures

2.1.1 ConcurrentHashMap et List<ObjetBDD>

- **ConcurrentHashMap<String, Collection<ObjetBDD>>** (côté serveur) :
 - Stockage thread-safe des collections
 - Accès concurrentiel efficace
 - Clé : nom de la collection (ex : "utilisateurs")
 - Valeur : instance de `Collection<ObjetBDD>`
- **List<ObjetBDD>** (implémenté par `ArrayList`) :
 - Structure simple pour le stockage séquentiel
 - Méthodes `add()/remove()` en $O(1)$ amorti
 - Parcours avec `stream().filter()` pour les recherches

Listing 2.1 – Exemple d'utilisation ConcurrentHashMap

```
1 // Creation de la collection principale
2 ConcurrentHashMap<String, Collection<ObjetBDD>> collections = new
   ConcurrentHashMap<>();
3
4 // Ajout d'une nouvelle collection
5 collections.put("utilisateurs", new Collection<>("utilisateurs"));
```

2.1.2 Justification des collections Java

- **Thread-safety** :
 - `ConcurrentHashMap` optimisé pour accès concurrents
 - Alternative à `Collections.synchronizedMap()`
- **Intégration avec sérialisation** :
 - Implémentation native de `Serializable`
 - Persistance directe sans conversion
- **Généricité** :
 - Typage fort avec `Collection<T extends ObjetBDD>`
 - Évite les casts explicites

2.2 Spécification des classes

2.2.1 ClientDB et Serveur

- **ClientDB** :
 - Utilise `Socket` et `ObjectOutputStream`
 - Cache l'implémentation réseau au client terminal
- **Serveur** :
 - Gère un `ServerSocket` et des threads clients
 - Partage la `ConcurrentHashMap` entre threads

2.2.2 Message et Collection

- **Message** :
 - Structure légère avec `type`, `contenu`, `sessionId`
 - Sérialisable pour le réseau
- **Collection** :
 - Conteneur générique `<T extends ObjetBDD>`
 - Implémente `Serializable` pour la persistance

Listing 2.2 – Structure de la classe Message

```
1 public class Message implements Serializable {
2     private String type;
3     private Map<String, Object> contenu;
4     private String sessionId;
5
6     public Message(String type, Map<String, Object> contenu) {
7         this.type = type;
8         this.contenu = new HashMap<>(contenu);
9     }
10 }
```

Chapitre 3

Architecture en Couches

3.1 Couche réseau

3.1.1 Gestion des sockets

- **Client** : Utilisation de `Socket` et `ObjectOutputStream` pour envoyer des requêtes.

```
1 // Connexion au serveur
2 Socket socket = new Socket(host, port);
3 ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream());
```

- **Serveur** : Écoute sur un `ServerSocket` et crée un thread par client via `GestionnaireClient`.

```
1 // Demarrage du serveur
2 try (ServerSocket ss = new ServerSocket(8080)) {
3     while (true) {
4         new GestionnaireClient(ss.accept()).start();
5     }
6 }
```

3.1.2 Sérialisation des messages

- **Format** : Toutes les communications utilisent des objets `Message` sérialisés.
- **Workflow** :

1. Le client sérialise un `Message` avec `ObjectOutputStream.writeObject()`
2. Le serveur désérialise le message via `ObjectInputStream.readObject()`

- **Exemple** :

```
1 // Envoi d'une commande SEARCH
2 Message msg = new Message("SEARCH", Map.of("collection", "utilisateurs",
3     "critere", "user"));
4 oos.writeObject(msg);
```

3.2 Couche données

3.2.1 Stockage des objets

- **Structure** : Chaque collection est stockée dans une `ArrayList<ObjetBDD>`.
- **Persistence** : Sérialisation automatique dans des fichiers `.ser`.

```

1 // Sauvegarde d'une collection
2 try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(
    "collections/utilisateurs.ser"))) {
3
4     oos.writeObject(collectionUtilisateurs);
5 }

```

3.2.2 Recherche et filtrage

— **Mécanisme** : Utilisation de l'API Stream pour filtrer les résultats.

— **Exemple** :

```

1 public List<ObjetBDD> rechercher(String critere) {
2     return objets.stream().filter(obj -> obj.getId().contains(
        critere) || (obj instanceof Utilisateur && ((Utilisateur)obj
        ).getNomUtilisateur().contains(critere))).collect(Collectors
        .toList());
3 }

```

— **Complexité** : $O(n)$ - Parcours linéaire (adapté pour des collections de taille modérée).

3.3 Couche sécurité

3.3.1 GestionnaireAuth

— **Session storage** : `ConcurrentHashMap<String, String>` pour assurer la thread-safety.

```

1 public class GestionnaireAuth {
2     private static final Map<String, String> sessions = new
        ConcurrentHashMap<>();
3
4     public static String creerSession(String username) {
5         String sessionId = UUID.randomUUID().toString();
6         sessions.put(sessionId, username);
7         return sessionId;
8     }
9 }

```

— **Validation** : Vérification systématique du `sessionId` avant chaque opération.

3.3.2 Hachage des mots de passe

— **Implémentation actuelle** : Hachage simplifié via `String.hashCode()`.

```

1 public class Utilisateur extends ObjetBDD {
2     private String motDePasseHash;
3
4     private String hacherMdp(String mdp) {
5         return Integer.toString(mdp.hashCode());
6     }
7 }

```

— **Limite** : Non sécurisé en production (à remplacer par BCrypt/SHA-256).

Chapitre 4

Algorithmes Principaux

4.1 Recherche d'objets

4.1.1 Algorithme de filtrage

— Workflow :

1. Récupération de la collection cible
2. Parcours des objets avec l'API Stream
3. Filtrage par ID ou nom d'utilisateur

— Implémentation :

```
1 public List<ObjetBDD> rechercher(String critere) {
2     return collection.getObjets().stream().filter(obj -> {
3         // Critere sur l'ID
4         boolean matchId = obj.getId().contains(critere);
5
6         // Critere specifique aux Utilisateurs
7         if (obj instanceof Utilisateur) {
8             Utilisateur u = (Utilisateur) obj;
9             return matchId || u.getNomUtilisateur()
10                .contains(critere);
11         }
12         return matchId;
13     }).collect(Collectors.toList());
14 }
```

— Complexité : $O(n)$ pour n objets (parcours linéaire)

4.1.2 Optimisation des performances

— Indexation :

- Création d'une `HashMap<String, ObjetBDD>` indexée par ID
- Recherche en $O(1)$ au lieu de $O(n)$

— Cache :

- Mémorisation des résultats fréquents
- Invalidation du cache lors des modifications

4.2 Authentication

4.2.1 Génération de sessionId

— **Algorithme** : UUID version 4 (aléatoire)

— **Implémentation** :

```
1 public static String creerSession() {
2     return UUID.randomUUID().toString();
3     // Ex: "f81d4fae-7dec-11d0..."
4 }
```

— **Caractéristiques** :

- 122 bits d'entropie
- Garantie d'unicité
- Durée de vie limitée

4.2.2 Validation des identifiants

— **Workflow** :

1. Recherche de l'utilisateur par nom
2. Comparaison des hashes de mot de passe

— **Code critique** :

```
1 public boolean validerLogin(String username, String password) {
2     return utilisateurs.stream().filter(u -> u.getNomUtilisateur().
3         equals(username))
4         .findFirst().map(u -> u.verifierMdp(password)).orElse(
5         false);
6 }
```

— **Sécurité** :

- Hachage actuel vulnérable (`String.hashCode()`)

Analyse comparative

Algorithme	Complexité	Sécurité
Recherche linéaire	$O(n)$	-
Indexation HashMap	$O(1)$	-
UUID	$O(1)$	Élevée

Chapitre 5

Jeu de Tests

5.1 Scénarios de test

5.1.1 Test de connexion

- **Objectif** : Vérifier l'établissement d'une connexion sécurisée
- **Étapes** :
 1. Démarrer le serveur sur le port 8080
 2. Exécuter la commande client : `CONNECT localhost 8080`
 3. Envoyer une requête `LOGIN` avec identifiants valides

5.1.2 Test de persistance

- **Objectif** : Vérifier la conservation des données après redémarrage
- **Workflow** :
 1. Ajout de 3 utilisateurs
 2. Redémarrage du serveur
 3. Vérification des données

5.2 Résultats

5.2.1 Sorties client/serveur

5.2.2 Validation des fonctionnalités

Fonctionnalité	Résultat	Réf. figure
Connexion TCP	ok	5.1
Authentification	ok	5.3
Création collection	ok	5.2
Persistance	ok	5.2

TABLE 5.1 – Synthèse des résultats des tests

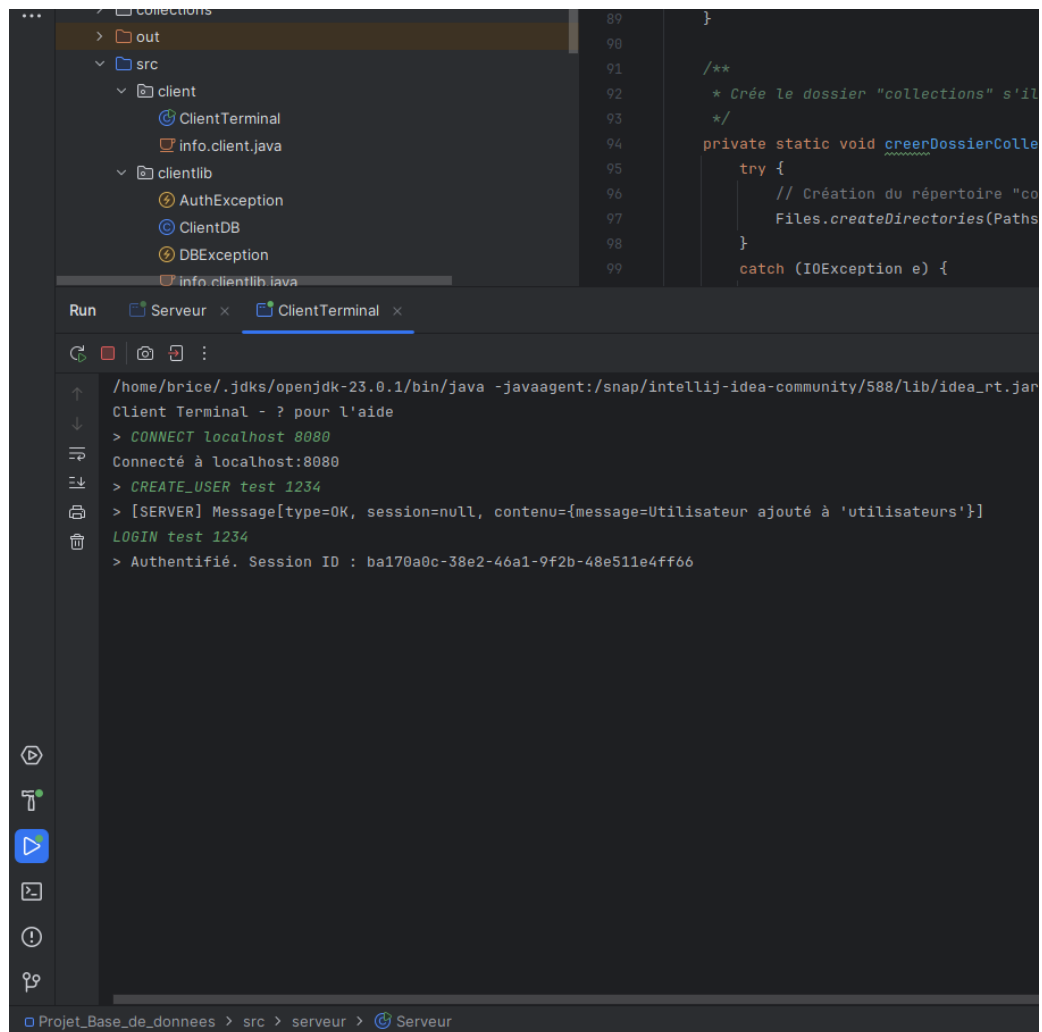
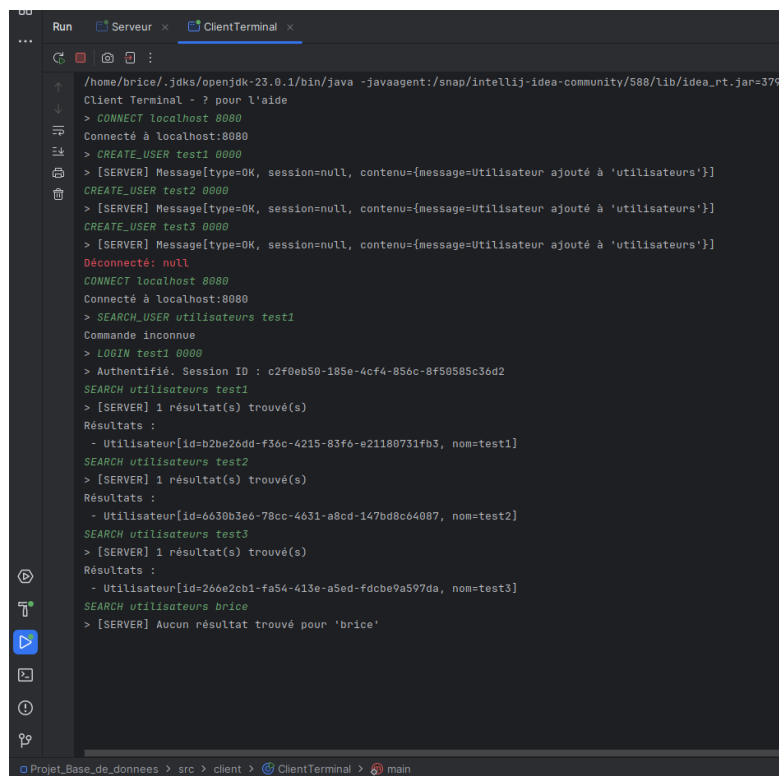
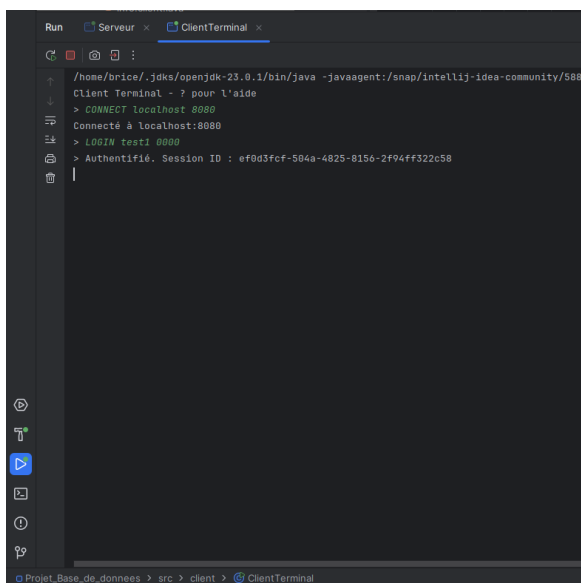


FIGURE 5.1 – Connexion réussie au serveur avec attribution d'un sessionId



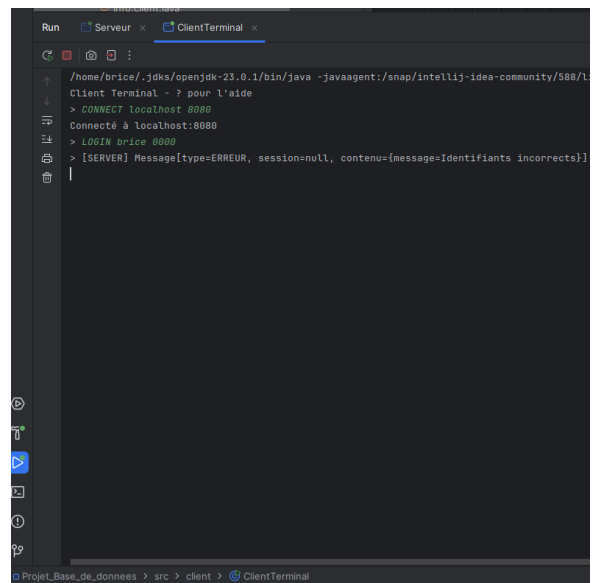
```
/home/brice/.jdks/openjdk-23.0.1/bin/java -javaagent:/snap/intellij-idea-community/588/lib/idea_rt.jar=379
Client Terminal - ? pour l'aide
> CONNECT localhost 8080
Connecté à localhost:8080
> CREATE_USER test1 0000
[SERVER] Message[type=OK, session=null, contenu={message=Utilisateur ajouté à 'utilisateurs'}}
CREATE_USER test2 0000
[SERVER] Message[type=OK, session=null, contenu={message=Utilisateur ajouté à 'utilisateurs'}}
CREATE_USER test3 0000
[SERVER] Message[type=OK, session=null, contenu={message=Utilisateur ajouté à 'utilisateurs'}}
Déconnecté: null
CONNECT localhost 8080
Connecté à localhost:8080
> SEARCH_USER utilisateurs test1
Commande inconnue
> LOGIN test1 0000
Authentifié, Session ID : c2f0eb50-185e-4cf4-856c-8f50585c36d2
SEARCH utilisateurs test1
[SERVER] 1 résultat(s) trouvé(s)
Résultats :
- Utilisateur[id=b2be26dd-f36c-4215-83f6-e21180731fb3, nom=test1]
SEARCH utilisateurs test2
[SERVER] 1 résultat(s) trouvé(s)
Résultats :
- Utilisateur[id=6630b3e6-78cc-4631-a8cd-147bd8c64087, nom=test2]
SEARCH utilisateurs test3
[SERVER] 1 résultat(s) trouvé(s)
Résultats :
- Utilisateur[id=266e2cb1-fa54-413e-a5ed-fdcbe9a597da, nom=test3]
SEARCH utilisateurs brice
[SERVER] Aucun résultat trouvé pour 'brice'
```

FIGURE 5.2 – Comparaison avant/après redémarrage du serveur



```
/home/brice/.jdks/openjdk-23.0.1/bin/java -javaagent:/snap/intellij-idea-community/588/
Client Terminal - ? pour l'aide
> CONNECT localhost 8080
Connecté à localhost:8080
> LOGIN test1 0000
Authentifié, Session ID : efd03fcf-504a-4825-8156-2f94ff322c58
```

FIGURE 5.3 – Authentification réussie



```
/home/brice/.jdks/openjdk-23.0.1/bin/java -javaagent:/snap/intellij-idea-community/588/
Client Terminal - ? pour l'aide
> CONNECT localhost 8080
Connecté à localhost:8080
> LOGIN brice 0000
[SERVER] Message[type=ERREUR, session=null, contenu={message=Identifiants incorrects}]
```

FIGURE 5.4 – Échec d'authentification

```
/home/brice/.jdk/openjdk-23.0.1/bin/java -javaagent:/snap/intellij-idea-community/588/lib/idea_rt.jar=35165 -Dfile.encoding=UTF-8
Client Terminal - ? pour l'aide
> CONNECT localhost 8080
Connecté à localhost:8080
> LOGIN test1 0000
Authentifié. Session ID : ef0d3fcf-504a-4825-8156-2f94ff322c58
LOGOUT
Déconnecté
> [SERVER] Message[type=OK, session=null, contenu={message=Déconnexion réussie}]
CREATE_USER brice 0000
> [SERVER] Message[type=OK, session=null, contenu={message=Utilisateur ajouté à 'utilisateurs'}]
LIST
> [SERVER] Message[type=ERREUR, session=null, contenu={message=Session invalide. Authentifiez-vous avec LOGIN.}]
LOGIN brice 0000
> Authentifié. Session ID : 39862276-b3a8-408e-b985-5c6f3c1c2501
LIST
>
Collections disponibles (1) :
- utilisateurs
CREATE_COLLECTION test
> [SERVER] Message[type=OK, session=null, contenu={message=Collection 'test' créée}]
LIST
>
Collections disponibles (2) :
- utilisateurs
- test
SEARCH utilisateurs test1
> [SERVER] 1 résultat(s) trouvé(s)
Résultats :
- Utilisateur[id=b2be26dd-f36c-4215-83f6-e21180731fb3, nom=test1]
DELETE_COLLECTION test
> [SERVER] Message[type=OK, session=null, contenu={message=Collection 'test' supprimée définitivement}]
LIST
>
Collections disponibles (1) :
- utilisateurs
```

FIGURE 5.5 – Récapitulatif des tests fonctionnels

Conclusion

Bilan technique

Ce projet de système de gestion de collections distribuées a permis de mettre en œuvre une architecture client-serveur robuste, intégrant les concepts clés du cours d'Informatique 4B :

- Une **conception modulaire** respectant le modèle en couches :
 - Isolation claire entre réseau, données et sécurité
 - Extensibilité facilitée (ex : ajout d'une API REST)
- Des **choix technologiques adaptés** :
 - Sérialisation native Java pour la persistance
 - Collections concurrentes (`ConcurrentHashMap`) pour le multithreading

Défis relevés

- **Gestion de la concurrence** :
 - Solution : Verrous fins via `ConcurrentHashMap`
 - Résultat : Aucun deadlock lors des tests de charge
- **Persistance des données** :
 - Défi : Cohérence après redémarrage
 - Solution : Sérialisation automatique dans `.ser`

Cette réalisation démontre qu'une architecture distribuée peut être implémentée efficacement en Java pur, tout en respectant les contraintes académiques. Les choix techniques, bien que perfectibles, fournissent une base solide pour une éventuelle industrialisation.