Rapport de projet - Intergiciels

Table des matières

Architecture globale

Diagramme de séquence type

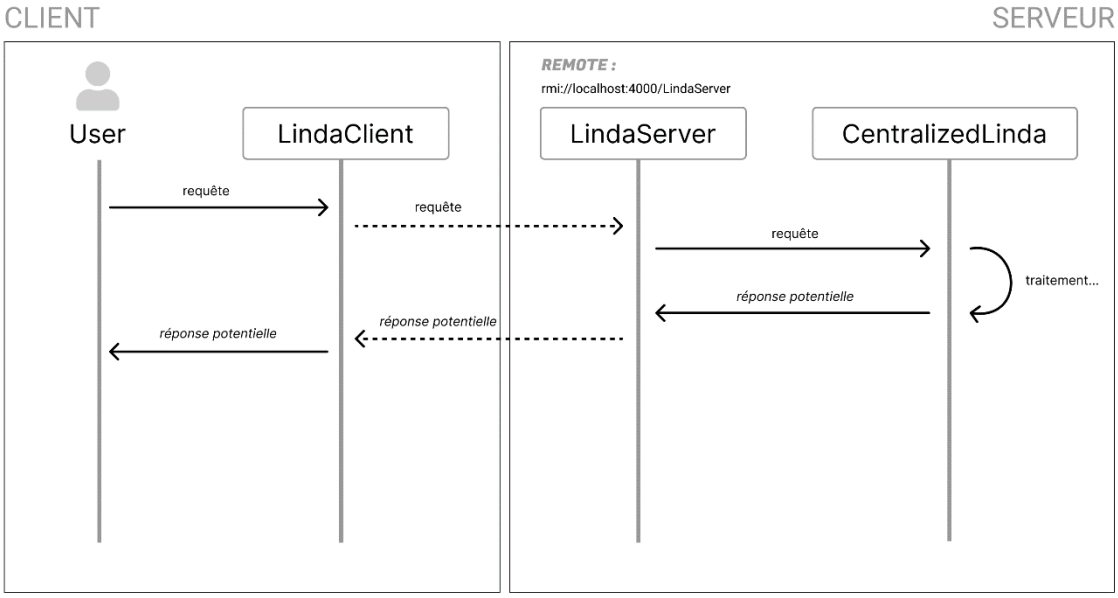
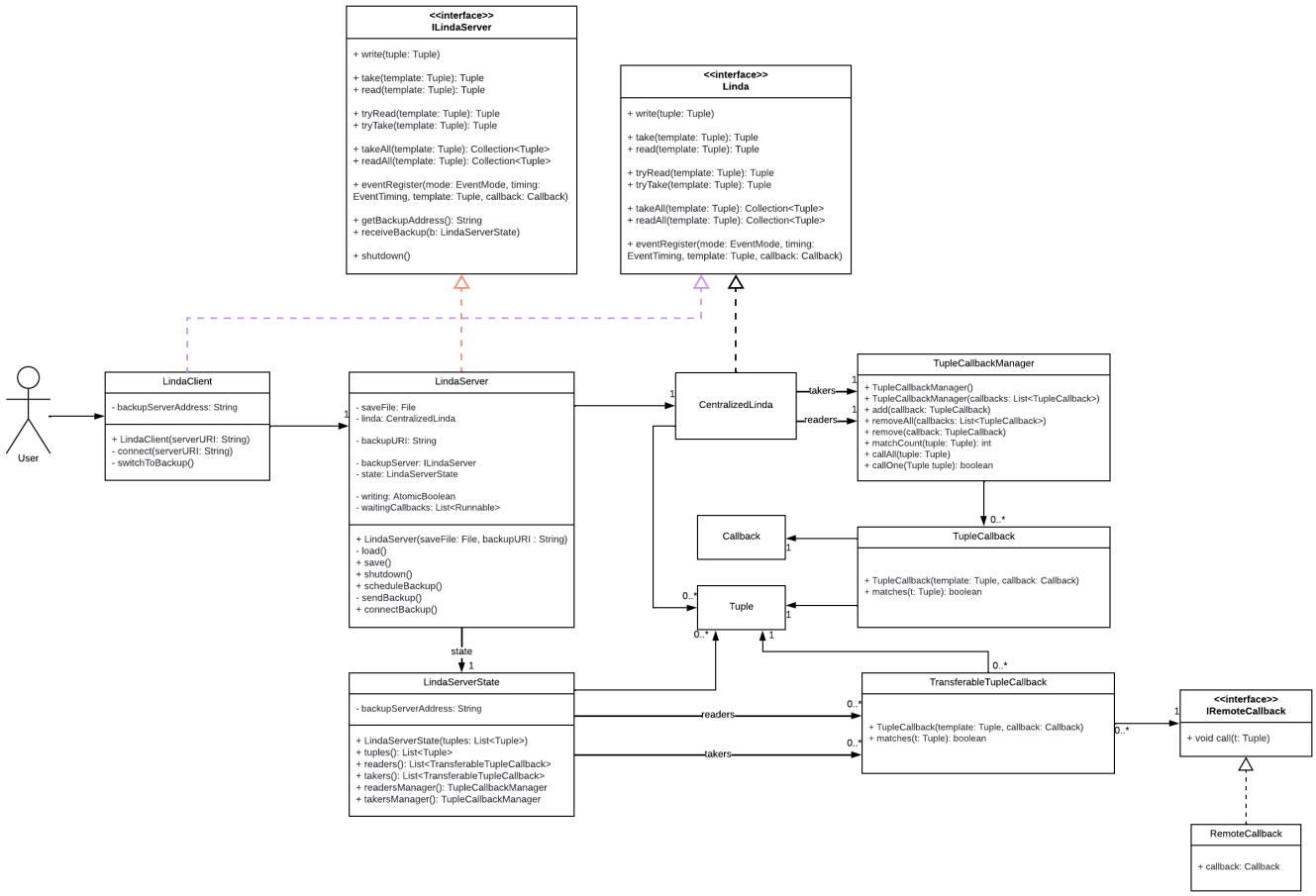


Diagramme de classes



Linda Centralisé

Pour réaliser le modèle Linda en version centralisée (`CentralizedLinda`) nous avons choisi de nous appuyer sur les forces de Java 8 en termes de traitement sur les collections (via l'API `Stream`) et de gestion de l'asynchrone (API `CompletableFuture`) pour vous proposer une solution moderne et simple, les deux susdits outils étant particulièrement adaptés pour ce projet (notamment grâce à la possibilité de se bloquer sur les `CompletableFuture`).

Nous avons opté pour la version "full callback" en représentant les `take` et `read` en attente par des `Callback`.

L'exclusion mutuelle est réalisée en annotant les méthodes avec `synchronized`.

Nos choix

- Pas de distinguo entre callback et appel direct des méthodes `read` et `take`.
- `Readers` appelé en premier lieu.
- `Taker` appelé ensuite.
- Politique FIFO.

Explications

Opérations synchrones : **`tryRead`**, **`tryTake`**, **`readAll`**, **`takeAll`**

Ces méthodes étant synchrones, leur fonctionnement est élémentaire.
Elles sont toutes exécutées en exclusion mutuelle (mot-clé `synchronized`) pour éviter un potentiel `write` ou `take` simultané qui interagirait sur la liste des tuples au mauvais moment.
Pour les méthodes `tryRead` et `tryRead` on récupère simplement une copie du premier Tuple qui matche le template indiqué, on le supprime dans le cas du `take`, puis on le retourne. S'il n'y en a pas on retourne `null`.

```
Optional<Tuple> tuple = this.tuples.stream().filter(t ->
t.matches(template)).findFirst().map(Tuple::deepclone);
if(!tuple.isPresent()) {
    return null;
}
if(take) {
    this.tuples.remove(tuple.get());
}
return tuple.get();
```

Pour les méthodes `readAll` et `takeAll`, c'est le même principe sauf qu'on récupère (et supprime si `take`) les copies de tous les tuples qui matchent le template pour les renvoyer (collection vide si aucun ne matche).

```
List<Tuple> tuples = this.tuples.stream()
                .filter(t ->
t.matches(template)).map(Tuple::deepclone).collect(Collectors.toList());
if(take) {
    this.tuples.removeAll(tuples);
}
return tuples;
```

La base des opérations asynchrones : eventRegister

Le fonctionnement de cette méthode est très simple.
Si le timing est IMMEDIATE alors on exécute l'équivalent d'un `tryRead` ou `tryTake` en fonction du mode. Si on trouve d'ores-et-déjà un tuple on appelle directement le callback avec le tuple trouvé et c'est terminé.
Sinon on crée un nouveau `TupleCallback` à partir du template et callback fournis, et on l'ajoute soit aux gestionnaires (`TupleCallbackManager`) des `readers`, soit à celui des `takers` (expliqués par la suite).
Un `TupleCallback` est un `CompletableFuture` qui, une fois complété, appelle le callback passé à son constructeur. Il contient aussi le template qu'il doit matcher pour pouvoir s'exécuter.

```
if (timing == eventTiming.IMMEDIATE) {
  Tuple tuple = get(template, mode == eventMode.TAKE);

  if (tuple != null) {
      callback.call(tuple);
      return;
  }
}

if (mode == eventMode.READ) {
  readers.add(new TupleCallback(template, callback));
} else {
  takers.add(new TupleCallback(template, callback));
}
```

Les opérations bloquantes **`read`** et **`take`**

Ces méthodes utilisent la force des `CompletableFuture` à pouvoir se bloquer en attente d'une valeur. Leur fonctionnement est ainsi extrêmement simple :

1. On crée un nouveau `CompletableFuture` acceptant un Tuple.
2. On enregistre un nouveau callback de mode `READ` ou `TAKE` avec un timing IMMEDIATE qui exécutera la méthode complete du `CompletableFuture` tout justement créé.
3. On retourne le résultat du `CompletableFuture` une fois complété.

Cela se fait en 4 lignes de code :

```
eventMode mode = take ? eventMode.TAKE : eventMode.READ;
CompletableFuture<Tuple> future = new CompletableFuture<>();
eventRegister(mode, eventTiming.IMMEDIATE, template, future::complete);
return future.join();
```

L'opération `write`

Encore une fois une méthode au fonctionnement extrêmement simple (décidément) :

```
public void write(Tuple t) {
	t = t.deepclone();

	// On débloque les readers correspondant
	readers.callAll(t);

	// On débloque un taker correspondant
	boolean taken = takers.callOne(t);

	synchronized(this) {
		if(!taken) {
		    tuples.add(t);
		}
	}
}
```

1. En premier lieu on crée une copie du tuple à écrire pour éviter que l'utilisateur puisse le modifier parallèlement à l'appel.
2. Ensuite le gestionnaire des `readers` appelle tous les callbacks qui matchent le tuple écrit.
3. Le gestionnaire des `takers` appelle le premier callback qui matche le tuple écrit (car un seul peut prendre ce nouveau tuple).
4. Si le tuple n'a pas été pris on l'ajoute à la liste des tuples pour qu'il puisse être lu et pris dans le futur.

Toute la magie se cache ici dans les fameux "gestionnaires" : `TupleCallbackManager`
Cette classe représente une liste de `TupleCallback` (vu précédemment). Les méthodes intéressantes sont `callAll(Tuple)` et `callOne(Tuple)`.

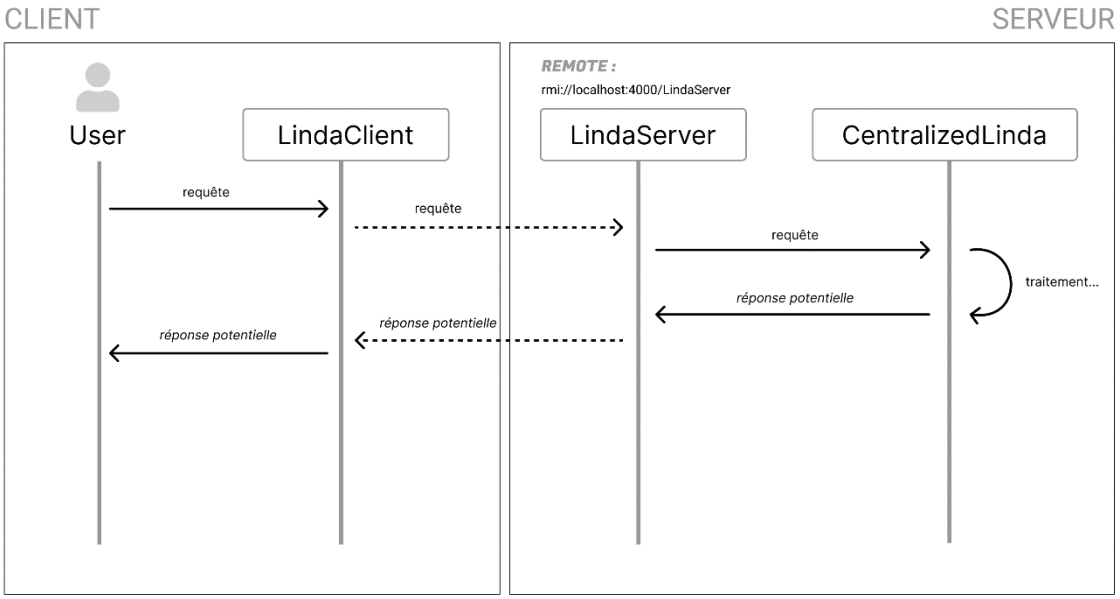
- `callAll(Tuple t)` complète tous les `TupleCallback` qui matchent le tuple t avec une copie de ce tuple, attend paisiblement que les threads en attente aient repris leur cheminement (méthode `CompletableFuture.allOf`), puis les supprime (conformément à la spécification).
- `callOne(Tuple t)` suit le même principe à l'exception près qu'on ne récupère que le premier TupleCallback qui matche t, et qu'on renvoie true si on a trouvé un ou false sinon.

On voit encore une fois la force des `CompletableFuture` qui permettent non seulement de se débloquer en recevant une valeur, mais aussi d'attendre qu'un ensemble de `CompletableFuture` aient été débloqués pour continuer à avancer.

Linda Client / Mono-serveur

Pour la version client / mono-serveur de Linda on a cherché à réutiliser le Linda centralisé créé précédemment pour séparer au maximum les responsabilités. Ici on ne gère donc que les interactions client / serveur et serveur principal / serveur de backup. Tout se fait par appel de méthode à distance (RMI) car selon nous c'est la méthode de transmission la plus simple dans ce cas.

Explications



Dans cette version tout se base sur la classe LindaServer accessible à distance. Cette classe ne fait en gros que rediriger les appels à distance vers le Linda centralisé interne qu'elle contient. A l'exception près de la gestion des callbacks et de la prévention des interblocages dû aux callbacks qui se s'auto-réenregistrent.

Le LindaClient contient l'instance du LindaServer accessible à distance, obtenue depuis le Registry à partir de l'adresse fournie dans son constructeur. Cette classe est également très simple et ne fait que rediriger les appels vers le LindaServer remote.

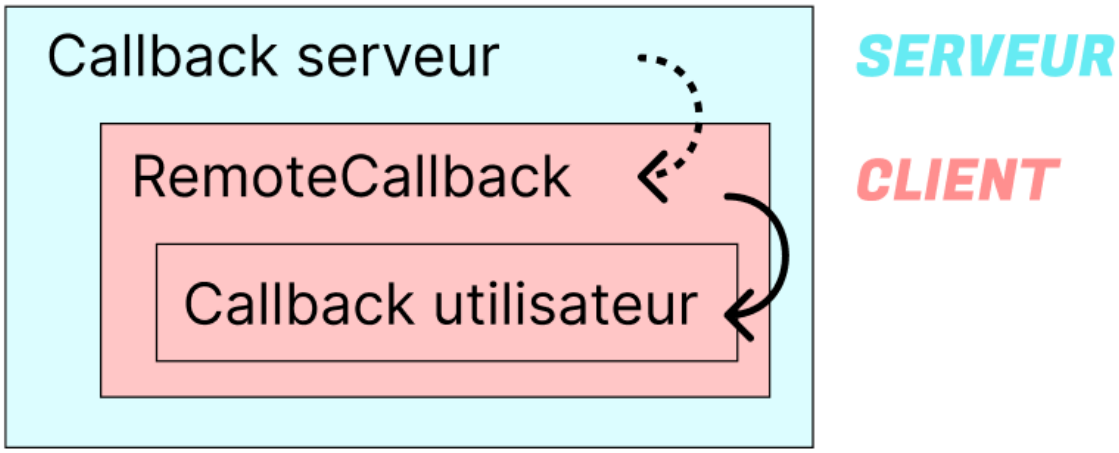
Gestion des callbacks

La gestion des callbacks est la partie acrobatique de cette implémentation. En effet les callbacks doivent être appelés côté client ; il faut donc trouver un moyen pour que l'exécution du callback côté serveur déclenche l'exécution du callback côté client.
Pour cela le moyen le plus simple est de créer un nouveau type de callback accessible à distance par le serveur. C'est donc une relation bidirectionnelle : le client accède au serveur à distance, et le serveur accède à distance aux callbacks des clients pour les exécuter.
Les callbacks par défaut n'étant pas accessible à distance il a fallu créer un nouveau type de callback `IRemoteCallback`.

Voici les différentes étapes qui permettent de faire fonctionner les callbacks :

1. Lorsque `LindaClient#eventRegister` est appelé le callback passé est enveloppé dans un `RemoteCallback` qui est ensuite envoyé au serveur.
2. Lorsque le serveur le reçoit, il l'enveloppe dans un Callback classique qu'il transmet au Linda centralisé.

Ainsi lorsque le callback du Linda centralisé est appelé il appelle le Callback à distance, lequel appelle le callback de l'utilisateur.



Mécanisme de sauvegarde

On cherche à sauvegarder les tuples afin de pouvoir redémarrer le serveur Linda en conservant son état. On suppose qu'il y a un mécanisme par dessus pour arrêter proprement le serveur par appel de sa méthode `shutdown`, ce système n'est pas là pour prévenir les crash ou éteintes sales. Egalement pour ne pas complexifier la solution on ne cherche pas à sauvegarder les callbacks car ils seront potentiellement caduques au moment de la reprise.

Le mécanisme de sauvegarde et de récupération est très simple :

- **Sauvegarde (méthode `save`)**
  On crée le fichier de sauvegarde s'il n'existe pas.
  On écrit la liste des tuples du Linda centralisé via un `ObjectOutputStream`.

```
try (ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream(saveFile))) {
    oos.writeObject(linda.getTuples());
} catch (IOException e) {
    e.printStackTrace();
}
```

- **Récupération**
  Le LindaServer attend le fichier de sauvegarde en paramètre.
  S'il existe on récupère l'état directement à l'appel du constructeur grâce à un `ObjectInputStream` :

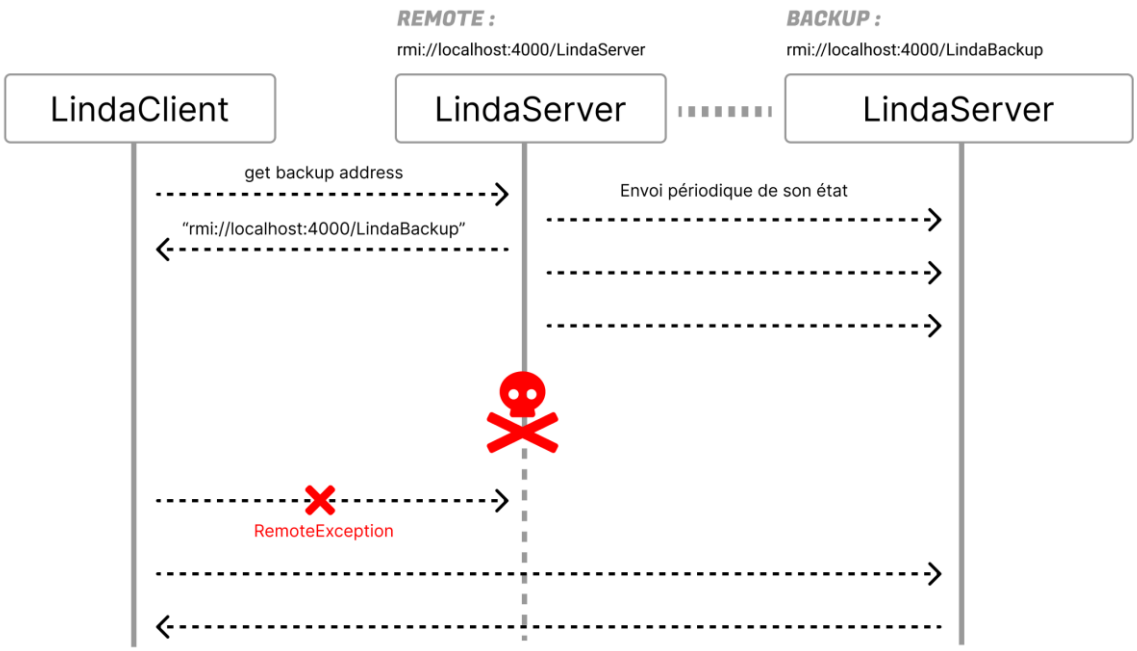
```
try (ObjectInputStream ois = new ObjectInputStream(new
FileInputStream(saveFile))) {
    return (List<Tuple>) ois.readObject();
} catch (IOException | ClassNotFoundException e) {
    e.printStackTrace();
}
System.err.println("Invalid save file.");
return new ArrayList<>();
```

  Puis on crée le Linda centralisé avec les tuples obtenus :

```
this.linda = new CentralizedLinda(tuples);
```

Bascule sur le serveur de backup

On souhaite gérer le dysfonctionnement du serveur Linda de manière transparente pour l'utilisateur en le faisant basculer automatiquement sur un serveur de backup.
On ne veut donc pas que l'utilisateur ait à spécifier lui-même l'adresse du serveur de backup. Pour cela on a une méthode `getBackupAddress` sur `ILindaServer` qui renvoie l'adresse du serveur de backup (ou `null` si aucun).
Le LindaClient connaît ainsi le backup, il gère ensuite la bascule automatiquement dès lors qu'une `RemoteException` est levée dans une méthode. En effet, s'il y a `RemoteException` c'est qu'il y a un problème avec le serveur alors on bascule.
Toutes les 5 secondes le serveur Linda principal envoie son état complet (`LindaServerState`) au serveur de backup, qui le reconstruit. On a supposé que les write et read / take étaient à peu près équilibrés et donc que le coup de ces transferts ne serait pas trop important.
Pour transférer les callbacks des utilisateurs le serveur principal a besoin de les stocker en interne dans des `TransferableCallback` contenant le template à matcher et le `IRemoteCallback` de l'utilisateur.



Exemple

```
Registry dns = LocateRegistry.createRegistry(4000);

ILindaServer backup = new LindaServer(new File("backup.bin"));
dns.bind("LindaBackup", backup);

ILindaServer linda = new LindaServer(new File("linda_data.bin"),
"rmi://localhost:4000/LindaBackup");
dns.bind("LindaServer", linda);
```

Simplement à titre d'illustration. Pour que cela ait un intérêt il faut bien-sûr mettre le registry, le serveur principal et le serveur de backup sur des machines différentes.

Plan de test

test.shm

Les classes de ce package visent à tester le linda centralisé.
`SelfRegisterInDifferentThread` : Teste qu'un callback puisse se réenregistrer même dans un thread différent. (Nous avions au départ un problème avec ça car les opérations `write` et `eventRegister` étaient en exclusion mutuelle, si le callback était appelé dans le même thread cela fonctionnait (le thread ayant déjà le lock), mais s'il était appelé dans un autre thread cela bloquait (cas du Whiteboard).

- `TestReadAll` : Teste que readAll renvoient bien les tuples escomptés (sans les retirer).
- `TestReadBeforeTake` : Teste que les read sont bien appelés avant les `take`
- `TestReadCallbackBeforeTake` : Teste que les callbacks `read` sont bien appelés avant les `take`
- `TestTakeReadAll` : Teste que `readAll` et `takeAll` fonctionnent bien même s'il y a des duplicats.
- `TestWriteCallback` : Teste de callback qui se réenregistre lui-même (compte le nombre d'appel).

test.server

- `ServerCallbackTest` : Teste que les callbacks fonctionnent sur serveur.
- `ServerWriteReadTest` : Teste que les `read` et `write` fonctionnent sur serveur.
- `TestBackup` : Teste que l'opération `getBackupAddress` fonctionne.
- `TestBackupServer` : (pas de junit) Teste que le serveur de backup prend bien le relai des callbacks.

```
Lancer linda.server.StartBackup
Lancer linda.server.StartServer
Lorsque "kill main" est affiché : tuer le serveur principal (StartServer)
Vérifier que le callback est quand-même appelé
```

- `TestRemoteCallbackSerializable` : Teste que les backup transférés entre serveur principal et serveur de backup sont bien sérialisables.