Tests unitaires de méthodes utilisant le réseau

1 Problématique

Faire les tests unitaires de méthodes utilisant le réseau est problématique. Comment tester si les données envoyées le sont correctement quand il n'y a pas un distant pour les recevoir. Pour tester la méthode d'envoi, on doit écrire des méthodes de réception. Mais si le test échoue, est-ce à cause des méthodes de réception écrites pour le test, ou à cause de celles d'envoi que l'on veut tester ? Et puis il y a ensuite le sens inverse à tester.

Et finalement, on se retrouve avec des tests unitaires qui utilisent en fait le code d'autres méthodes que l'on voudrait tester unitairement aussi.

Pour résoudre ce problème, il faudrait sur le réseau des services de tests qui seraient prêts à envoyer des données pour tester nos réceptions et à recevoir nos données pour tester nos émissions. Nous n'avons pas ce type de services à vous proposer pour vos tests.

Mais pour résoudre ce problème nous vous proposons un code qui devra être considéré comme digne de confiance [1] et qui vous permet de tester vos méthodes une par une, sans utiliser le code de l'autre.

Ce code vous est proposé sous la forme d'une bibliothèque qui vous permet d'activer certain services durant vos tests. Voici quelques exemples de services possibles :

— activer un serveur TCP qui renvoie en «écho» tous les octets qu'il reçoit ;
— activer un client TCP qui va sauver sur un fichier de votre arborescence tous les octets qu'il reçoit ;
— activer une alarme qui va interrompre votre programme si celui-ci est bloqué sur une entrée/sortie trop longtemps.

2 Installation de la bibliothèque

Cette bibliothèque se présente sous la forme d'une archive JAR à intégrer à votre projet.

Pour cela il faut modifier la configuration de Maven de votre projet. Comme cette bibliothèque n'est pas sur les dépôts standards il faut ajouter une section `<repositories>` avec l'adresse du dépôt, et ensuite ajouter une dépendances pour les tests. Les lignes à ajouter à votre fichiers *pom.xml* sont sur le listing de la page suivante.

Une fois ces lignes ajoutées, il faut rendre cohérent votre projet avec les commandes :

```
1 mvn clean install
2 mvn eclipse:clean eclipse:eclipse
```

La seconde commande régénère les fichiers de configuration d'Eclipse pour votre projet. Il vaut mieux avoir arrêter Eclipse avant de la lancer.

1. comme il a été écrit tout récemment et qu'il n'a pas été trop testé lui même, cette confiance est peut-être encore mal placée !

```
<project>
    (...)
  <repositories>
    <repository>
      <id>csc4509-dm-stable</id>
      <name>CSC4509, DM library, stable</name>
<url>http://www-inf.telecom-sudparis.eu/COURS/CSC4509/maven-repository/stable/</url>
    </repository>
    <repository>
      <id>csc4509-dm-snapshot</id>
      <name>CSC4509, DM library, snapshot</name>
<url>http://www-inf.telecom-sudparis.eu/COURS/CSC4509/maven-repository/snapshot/</url>
    </repository>
  </repositories>
  (...)
  <dependencies>
     (...)
    <dependency>
      <groupId>eu.telecomsudparis.csc4509</groupId>
      <artifactId>csc4509-dm-TestTCP</artifactId>
      <version>1.0-SNAPSHOT</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

3 Liste des services

La liste des services est disponible à cette URL :

http://www-inf.telecom-sudparis.eu/COURS/CSC4509/maven-repository/site/apidocs/tsp/csc4509/tcpservices/TcpTestTools.htm

Tous ces services journalisent le début et la fin de leur activité sur le `Log.TEST` au niveau `TRACE`. En cas de fin prématurée à cause d'une exception, le message d'exception est journalisé au niveau `DEBUG`.

TcpTestTools() : il s'agit du constructeur de la classe. Il permet d'obtenir une référence sur une instance de l'outil pour appeler tous les services. Un seul exemplaire de l'outil suffit pour votre classe de test, il est donc judicieux de créer cette instance dans la méthode (static) annotée *@BeforeAll*. Voir l'exemple dans la section suivante de cette documentation ;

arlarmStart(long timeout) : débute un service qui envoie une interruption à votre thread au bout de *timeout* millisecondes s'il n'est pas stoppé avant. Ce service est utile si vous testez une méthode qui peut être bloquante. Si la scénario se passe mal, la méthode bloque et les tests ne se terminent jamais. Ce service stoppe le test en avec l'exception `InterruptedException`. Si le test ne prévoie pas cette exception, il passe en échec.

arlarmStop() : stoppe l'alarme en cours, et évite que l'interruption ne soit envoyée. Ne fait rien s'il n'y a aucune alarme en cours.

clientStart(int port) : débute un client `TCP` qui se connecte au serveur *locahost :port* et s'arrête automatiquement si la connexion a réussi.

clientStop() : stoppe le client `TCP` lancé par *clientStart()*. N'est utile que si la connexion n'a pas été établie puis fermée. Ne fait rien si le client est déjà terminé.

clientReceiveFileStart(int port, String fileName) : débute un client `TCP` qui se connecte au serveur *locahost :port* et qui sauvegarde tous les octets reçus dans le fichier *fileName* qui est écrasé à son ouverture. Les données ne sont intégralement écrites dans le fichier qu'à la fermeture de la connexion. Un arrêt par *clientReceiveFileStop()* ne garantit pas l'écriture de tous les octets reçus dans la file `TCP`. Ce client s'arrête de lui même en cas de fermeture de la connexion.

clientReceiveFileStop() : stoppe le client lancé par *clientReceiveFileStart()*. N'est utile que si la connexion n'a pas été fermée. Ne fait rien si le client est déjà terminé.

clientSendFileStart(int port, String fileName) : débute un client `TCP` qui se connecte au serveur *locahost :port* et qui lui envoie le fichier *fileName*. Il ferme ensuite la connexion et se termine de lui même.

clientSendFileStop() : stoppe le client lancé par *clientSendFileStart()*. N'est utile que si la connexion n'a pas été fermée. Ne fait rien si le client est déjà terminé.

echoStart(int port) : débute un serveur `TCP` sur l'adresse *localhost :port* qui retourne en «echo» tous les octets que lui envoie le premier client qui se connecte sur le port passé en paramètre. Le service se termine de lui même à la fermeture de connexion.

echoStop() : stoppe le serveur `TCP` lancé par *echoStart()*. N'est utile que si la connexion n'a pas été fermée. Ne fait rien si le serveur est déjà terminé.

serverReceiveFileStart(int port, String fileName) : débute un serveur `TCP` sur l'adresse *localhost :port* qui sauvegarde tous les octets envoyés par le premier client connecté dans le fichier *fileName*. Ce fichier est écrasé à son ouverture. Les données ne sont intégralement écrites dans le fichier qu'à la fermeture de la connexion. Un arrêt par *serverReceiveFileStop()* ne garantit pas l'écriture de tous les octets reçus dans la file `TCP`. Ce serveur s'arrête de lui même en cas de fermeture de la connexion. Ce service fait une pause d'une seconde pour laisser au thread du serveur le temps d'agir.

serverReceiveFileStop() : stoppe le serveur lancé par *serverReceiveFileStart()*. N'est utile que si la connexion n'a pas été fermée. Ne fait rien si le client est déjà terminé.

serverSendFileStart(int port, String fileName) : débute un serveur `TCP` sur l'adresse *localhost :port* qui envoie le fichier *fileName* au premier client qui se connecte. Il ferme ensuite la connexion et se termine de lui même. Ce service fait une pause d'une seconde pour laisser au thread du serveur le temps d'agir.

serverSendFileStop() : stoppe le serveur lancé par *serverSendFileStart()*. N'est utile que si la connexion n'a pas été fermée. Ne fait rien si le client est déjà terminé.

serverStart(int port) : débute un serveur `TCP` sur l'adresse *localhost :port* qui accepte des clients en boucle infinie. Le serveur ne lit aucune des données émises par les clients (qui peuvent donc être bloqués en écriture en cas de congestion des files d'attente `TCP`). Ce service ne peut être arrêté que par l'appel de *serverStop()*. Ce service fait une pause d'une seconde pour laisser au thread du serveur le temps d'agir.

serverStop() : stoppe le serveur lancé par *serverStart(int port)*. Ne fait rien si le serveur est déjà terminé.

4 Exemples d'usage

Voici quelques exemples d'usage pour les tests unitaires de votre projet. Les exemples de codes commentés ci dessous sont déjà écrits dans les classes de test de l'étape 2 du DM :

https://www-inf.telecom-sudparis.eu/COURS/CSC4509/DM/DM2/TestTcpServer.java

et

https://www-inf.telecom-sudparis.eu/COURS/CSC4509/DM/DM2/TcpSocket.java

4.1 Lancement du service

Le lancement du service se fait dans une méthode static annotée «*@BeforeAll*» de votre classe de test. Profitez de cette méthode pour régler le niveau de `Log`. Au niveau par défaut du `Log.TEST` tous ces services sont muets sauf en cas d'exception anormale. Si vous voulez suivre la trace des services, il faut baisser le niveau du `Log.TEST` au niveau `TRACE`.

```
1 @BeforeAll
2 static public void lanceTestTools() {
3     tcpTestTools = new TcpTestTools();
4     Log.configureALogger(LOGGER_NAME_TEST, Level.TRACE);
5 }
```

4.2 Test de la méthode TcpServer : :accept()

But du test : vérifier que la méthode *TcpServer : :accept()* de votre code permet bien l'établissement d'une connexion `TCP` lorsqu'un client tente de s'y connecter.

```
1 // test unitaire de l'accept du serveur.
2 @Test
3 public void testAccept() throws Exception {
4   TEST.info("TestTcpServer::testAccept -> test d'acceptation d'un client");
5
6   // Ici test unitaire du constructeur du serveur
7   TcpServer serveur = new TcpServer(SERVERPORT);
8   // si aucune exception n'est levé, le test OK.
9   tcpTestTools.alarmStart(5000); // on laisse 5 secondes pour que la connexion s'établis
10  tcpTestTools.clienStart(SERVERPORT); // on demande à un client de se connecter à notre
11
12  // Ici test unitaire de la méthode acceptClient()
13  TcpSocket client = serveur.acceptClient();
14  // on n'arrive ici que si il n'y pas eu d'exception. C'est donc un succès.
15  tcpTestTools.alarmStop(); // connexion ok. On stope l'alarme.
16  client.close();
17  // Ici test unitaire de la méthode close() du serveur
18  serveur.close();
19  // si aucune exception n'a été lévée, le test est OK.
20 }
```

ligne 7 : création d'un `TcpServer` en utilisant votre code (c'est ici un premier test de votre code) ;

ligne 9 : mise en place d'une alarme de `TcpTestTools` qui stoppe le test par une interruption si la connexion n'est pas établie au bout de 5 secondes. Une demande de connexion peut bloquer plusieurs minutes en cas de problème. Pour éviter de bloquer vos tests, cette alarme casse ce test au bout de 5 secondes. Il échoue alors à cause d'une exception ;

ligne 10 : lancement d'un client par `TcpTestTools` qui se connecte à votre serveur ;

ligne 13 : acceptation du client par votre code (c'est donc ici, un deuxième test de votre code) ;

ligne 15 : tout s'est bien passé, l'alarme est stoppée pour éviter de provoquer une erreur du test.

4.3 Test du constructeur de TcpSocket

But du test : vérifier que le constructeur *TcpSocket(String serverHost, int serverPort)* établie bien une connexion avec un serveur écoutant sur l'adresse `TCP/IP` désignée.

```
1 // test unitaire de la connexion d'un client
2 @Test
3 public void testTcpSocketClient() throws Exception {
4   TEST.info("[TestTcpSocket::testTcpSocketClient] -> test de création d'un client");
5
6   // lancement d'un serveur de test  TCP
7   tcpTestTools.serverStart(TESTSERVERORT);
8
9   // on laisse 5 secondes pour que la connexion s'établisse.
10  tcpTestTools.alarmStart(5000);
11
12  // connexion du client
13  TcpSocket client = new TcpSocket("localhost", TESTSERVERORT);
14  // si aucune exception n'est levé, le test OK.
15
16  client.close();
17  tcpTestTools.alarmStop();
18  tcpTestTools.serverStop();
19 }
```

ligne 7 : lancement par `TcpTestTools` d'un serveur `TCP` qui écoute sur l'adresse *localhost :TESTSERVERORT* ;

ligne 10 : mise en place d'une alarme de `TcpTestTools` qui stoppe le test par une interruption si la connexion n'est pas établie au bout de 5 secondes. Une demande de connexion peut bloquer plusieurs minutes en cas de problème. Pour éviter de bloquer vos tests, cette alarme casse ce test au bout de 5 secondes. Il échoue alors à cause d'une exception ;

ligne 13 : création d'un client `TCP/IP` sur l'adresse *localhost :TESTSERVERORT* avec votre code (votre code est testé ici) ;

ligne 17 : fin de l'alarme puisque la connexion est passée ;

ligne 18 : le serveur lancé par *TcpTestTools : :serverStar()* ne s'arrête pas de lui même. Pour éviter de continuer à utiliser l'adresse `TCP/IP` *localhost :TESTSERVERORT* , il faut le stopper.

4.4 Test de la méthode TcpSocket : :sendBuffer()

But du test : vérifier que la méthode *TcpSocket : :sendBuffer()* envoie bien des données et que ces données sont bien celles voulue.

Ce test présuppose la présence du fichier "./data/fichierTemoin.txt" contenant des données quelconques. Pour que le test soit plus fiable, veuillez à ce que la taille du fichier dépasse la taille du buffer de la boucle d'envoi (donc 1024 octets pour le code exemple ci dessous).

Principe du test : faire envoyer un fichier par votre code, le recevoir et le sauver par un serveur, et comparer les deux fichiers.

```
1  @Test
2  public void testSendBuffer(@TempDir Path tempDir) throws Exception {
3    final String RECEIVEFILENAME = tempDir.resolve("recu.bin").toString();
4    final String SENDFILENAME = tempDir.resolve("envoye.bin").toString();
5
6    TEST.info("[TestTcpSocket::testSendBuffer] -> test d'envoie d'un message avec un clien
7    // création d'un fichier aléatoire contenant les données à envoyer.
8    createDataTestFile(SENDFILENAME, 0);
9
10   // lancement d'un serveur de test TCP
11   tcpTestTools.serverReceiveFileStart(TESTSERVERORT, RECEIVEFILENAME);
12
13   // création d'un client
14   TcpSocket client = new TcpSocket("localhost", TESTSERVERORT);
15
16   FileInputStream fin = new FileInputStream(SENDFILENAME);
17   FileChannel fcin = fin.getChannel();
18   ByteBuffer buffer = ByteBuffer.allocate(BUFFERSIZE);
19   int lu;
20   do {
21     buffer.clear();
22     lu = fcin.read(buffer);
23     buffer.flip();
24     client.sendBuffer(buffer);
25   } while (lu == BUFFERSIZE);
26   client.close(); // attention, la lecture de l'autre coté ne débloque que
27   // si le buffer est plein, ou la connexion fermée.
28   // il faut donc fermer le client pour que le serveur termine sa lecture
29   // et sauve la fin du fichier.
30
31   Thread.sleep(1000); // on laisse un peu de temps au serveur
32   // pour qu'il termine sa réception avant de le stopper.
33   tcpTestTools.serverReceiveFileStop();
34   assertTrue( compareFile(RECEIVEFILENAME, SENDFILENAME),
35     "Données reçus identiques aux données envoyées");
36   fin.close();
37 }
```

ligne 8 : création d'un fichier temporaire qui contient les données qui seront envoyées par la méthode à tester (sendBuffer());

ligne 11 : lancement par `TcpTestTools` d'un serveur `TCP` qui écoute sur l'adresse *localhost :TESTSERVERORT* et qui sauve dans le fichier *"./tmp/recu.txt"* tous les octets que le premier client connecté lui envoie ;

ligne 14 : création d'un client `TCP/IP` sur l'adresse *localhost :TESTSERVERORT* avec votre code (premier test de votre code ici) ;

lignes 16-25 : envoi du fichier *SENDFILENAME* (crée ligne 8) au serveur. La méthode *TcpSocket : :sendBuffer()* est testée ligne 24. Le serveur fonctionne sur la même logique que votre code, et donc il n'achève ses lectures que si son buffer est plein où si la connexion est fermée. Il faut donc penser à fermer la connexion (ligne 26) ;

ligne 31 : il faut laisser un tour d'exécution au thread du serveur pour qu'il reçoive et sauve les données. Le *Thread.sleep(1000)* est important pour que le thread du test fasse une pause avant la vérification des fichiers ;

ligne 34 : si le test est réussi le serveur de `TcpTestTools` est déjà terminé, mais en cas d'anomalie on le stoppe pour éviter de continuer d'utiliser l'adresse `TCP/IP` *localhost :TESTSERVERORT* ;

lignes 36 on vérifie que les données reçues par le serveur sont identiques à celles envoyées par votre code.

4.5 Test de la méthode TcpSocket : :sendObject()

But du test : vérifier que la méthode *TcpSocket : :sendObject()* envoie bien les données permettant de désérialiser l'objet.

Principe du test : faire envoyer un objet par votre code, le recevoir la trame par un serveur qui la sauve dans un fichier, lire ce fichier pour reconstruire l'objet et comparer l'objet reconstruit avec l'objet original.

```
1  @Test
2  public void testSendObject(@TempDir Path tempDir) throws Exception {
3    // nom du fichier ou l'objet␣serialisé␣va␣être␣sauvé.
4  ␣␣final␣String␣OBJECTFILENAME␣=␣tempDir.resolve("objet.data").toString();
5  ␣␣TEST.info("[TestTcpSocket::testSendObject]␣->␣␣test␣d'envoi d'un␣objet␣avec␣un␣client
6
7  ␣␣//␣lancement␣d'un serveur de test TCP
8    tcpTestTools.serverReceiveFileStart(TESTSERVERORT, OBJECTFILENAME);
9
10   // création d'un␣client
11 ␣␣TcpSocket␣client␣=␣new␣TcpSocket("localhost",␣TESTSERVERORT);
12
13 ␣␣//␣création␣d'un objet:
14   String str = "L'objet␣de␣test␣sera␣cette␣chaîne␣de␣caractères";
15
16   // test unitaire de la méthode sendObject()
17   client.sendObject(str);
18   client.close();
19
20   Thread.sleep(1000); // on laisse un peu de temps au serveur.
21   tcpTestTools.serverReceiveFileStop();
22
23   // l'objet␣serialisé␣doit␣être␣arrivé␣dans␣le␣fichier␣du␣serveur
24 ␣␣FileInputStream␣fin␣=␣new␣FileInputStream(OBJECTFILENAME);
25 ␣␣FileChannel␣fcin␣=␣fin.getChannel();
26
27 ␣␣//␣lecture␣de␣la␣taille␣en␣début␣de␣trame
28 ␣␣ByteBuffer␣bufferSize␣=␣ByteBuffer.allocate(Integer.BYTES);
29 ␣␣fcin.read(bufferSize);
30 ␣␣bufferSize.flip();
31 ␣␣int␣size␣=␣bufferSize.getInt();
32
33 ␣␣//␣lecture␣de␣l'objet sérialisé
34   ByteBuffer buffer = ByteBuffer.allocate(size);
35   fcin.read(buffer);
36   buffer.flip();
37
38   // désérialisation de l'objet
39 ␣␣ByteArrayInputStream␣bi␣=␣new␣ByteArrayInputStream(buffer.array());
40 ␣␣ObjectInputStream␣oi␣=␣new␣ObjectInputStream(bi);
41 ␣␣String␣strRecu␣=␣(String)␣oi.readObject();
42 ␣␣oi.close();
43 ␣␣bi.close();
44 ␣␣fin.close();
45
46 ␣␣assertTrue(str.equals(strRecu),"Objet␣reçu␣identique␣à␣l'objet envoyé");
47 }
```

ligne 8 : lancement par `TcpTestTools` d'un serveur `TCP` qui écoute sut l'adresse *localhost :TESTSERVERORT* et qui sauve dans le fichier *OBJECTFILENAME* tous les octets que le premier client connecté lui envoie ;

ligne 11 : création d'un client `TCP/IP` sur l'adresse *localhost :4001* avec votre code (testé ici) ;

lignes 13-18 : création d'on objet, et envoi de cet objet par la méthode *TcpSocket : :sendObject()* de votre code (testé ici) ;

ligne 20 : il faut laisser un tour d'exécution au thread du serveur pour qu'il reçoive et sauve les données. Le *Thread.sleep(1000)* est important pour que le thread du test fasse une pause avant la lecture des données de l'objet dans le fichier ;

ligne 21 : si le test est réussi le serveur de `TcpTestTools` est déjà terminé, mais en cas d'anomalie on le stoppe pour éviter de continuer d'utiliser l'adresse `TCP/IP` *localhost :TESTSERVERORT* ;

lignes 27-31 : lecture dans le fichiers des octets en débuts de trame qui contiennent la taille des données contenant l'objet sérialisé ;

lignes 33-36 : lecture de l'objet sérialisé,

lignes 38-43 : désérialisation de l'objet ;

ligne 46 : comparaison de l'objet sérialisé avec l'objet original.