UE INF4003 : TP 1
Développement d'un support de communication pour les processus séquentiels communicants (CSP)

Frédéric Raimbault

Rappel : ce TP doit être réalisé intégralement avant la séance, il sera testé et noté en début de séance. Au début de la séance, un travail complémentaire vous sera remis et vous serez évalué en temps limité sur ce nouveau travail. Les TP sont testés sous linux, sur les PC des salles de TP et sont exécutés sur le cluster *dmis*, à l'exclusion de toute autre machine ou système. Votre travail doit rester personnel et tout code partagé (que ce soit des scripts ou du code Java), repris d'un autre étudiant ou copié sur le web sera considéré comme une tentative de fraude et traité en conséquence.

1 Première partie : synchronisation manuelle

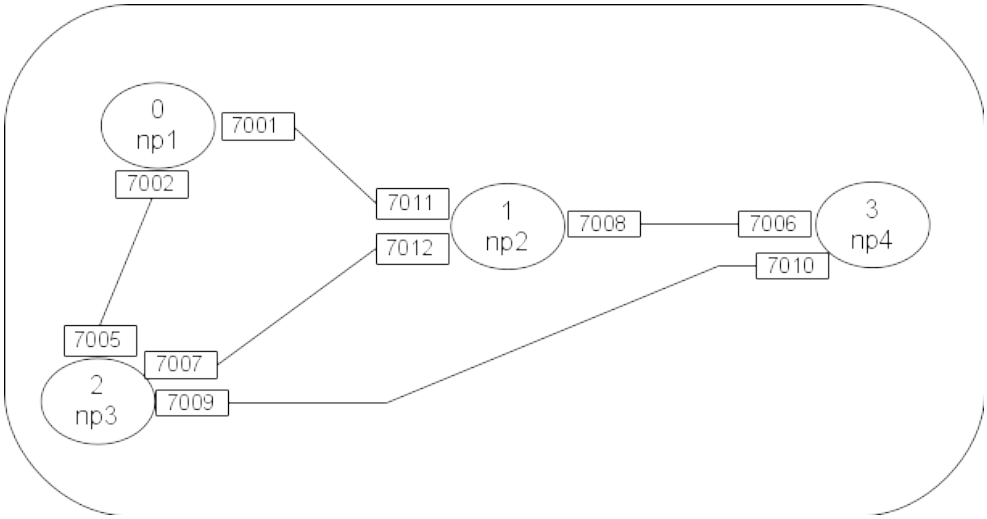
1.1 Bibliothèque de communication

L'objectif de ce TP est de construire une bibliothèque Java permettant la mise en œuvre rapide d'algorithmes distribués en suivant le modèle de programmation utilisé en cours. Dans ce modèle, les processus sont séquentiels, numérotés à partir de 0, les envois de messages (non bloquants) sont explicites et la réception d'un message déclenche un traitement associé à un tag. Chaque processus connaît uniquement son identité et celles de ses voisins.

La mise en œuvre de cette bibliothèque de communication repose sur des communications par sockets UDP Java. La topologie du réseau est simulée en contraignant les canaux de communications entre voisins. Un fichier décrit la topologie complète du réseau ; chaque ligne du fichier indique le canal de communication bi-directionnel existant entre un nœud et un de ses voisins : le nom de l'hôte (*hostname*), son numéro de processus, le port local, le nom de l'hôte du voisin, le numéro de processus du voisin, le port distant. Par exemple, la figure 1 illustre une représentation graphique d'un réseau et sa description textuelle. Toutes les éléments d'une ligne de description sont séparés par le symbole « : ».

Comme votre programme s'exécutera (sur les nœuds du cluster) potentiellement en même temps que ceux d'autres utilisateurs, vous devez faire en sorte de ne pas utiliser les mêmes ports que ceux des programmes des autres utilisateurs. C'est pourquoi à la création d'un processus vous donnerez une valeur d'offset qui sera systématiquement ajoutée aux

valeurs données dans le fichiers de configuration ; arrangez-vous pour utiliser une valeur d'offset qui vous réserve une plage de port différente les uns des autres.




```
np1 : 0 : 7001 : np2 : 1 : 7011
np1 : 0 : 7002 : np3 : 2 : 7005
np2 : 1 : 7012 : np3 : 2 : 7007
np2 : 1 : 7008 : np4 : 3 : 7006
np4 : 3 : 7010 : np3 : 2 : 7009
```

Fig. 1: Exemple de réseau et sa description textuelle

1.2 Mise en œuvre de la bibliothèque

La mise en œuvre de la bibliothèque est à réaliser dans le package Java `csp_V0` dont la Javadoc vous est fournie sur l'ENT. *Ce document doit être considérée comme une spécification à respecter scrupuleusement* : vous devez conserver les mêmes noms de packages, de méthodes et de champs, avec les mêmes attributs. Le source des classes `ThreadLoop` et `Neighbouring` vous sont fournis, il ne doivent pas être modifiés ; vous noterez que `ThreadLoop` contient une méthode de trace qu'il vous est conseillé d'utiliser pour la mise au point de votre bibliothèque. Le reste du package `csp_V0` est à écrire en vous aidant des explications qui suivent.

1.2.1 Processus

La classe centrale du package `csp_V0` est la classe abstraite `ConcurrentProcess` qui modélise un processus séquentiel communiquant (Concurrent Sequential Process). Ce processus crée un ensemble d'écouteurs (classe `MessageListener`) pour chaque canal de communication avec ses voisins. Ces deux classes héritent de la classe abstraite `ThreadLoop` qui modélise une tâche itérative dont on spécifie l'initialisation (méthode `beforeLoop()`),

le corps de l'itération (méthode `inLoop()`) et la fin d'exécution (méthode `afterLoop()`) ; la sortie de l'itération est déclenchée par l'appel à la méthode `exitLoop()`.

1.2.2 Voisinage

La structure de données modélisant le voisinage d'un processus est construite et mémorisée dans la classe `Neighbouring` à partir d'un fichier tel que décrit précédemment (méthode `Neighbouring.read()`). A noter que pour simplifier tous les processus lisent le même fichier mais que chacun ne mémorise que les informations utiles à l'établissement des liens de communication avec ses voisins.

Pendant la construction de leur voisinage, les processus créent leurs écouteurs (classe `MessageListener`) chargés de surveiller les ports d'entrées des canaux de communication et de déclencher le traitement associé à la réception des messages (appel à la méthode `ConcurrentProcess.receiveMessage()` décrite plus bas).

Notez que le code du processus et ceux des traitements de la réception des messages (cf. classe `MessageHandler` décrite ci-dessous) sont susceptibles de s'exécuter en parallèle. Il faut donc gérer la concurrence sur les structures de données partagées entre les threads, notamment en utilisant les moyens de synchronisations fournis par l'API Java (package `java.util.concurrent`) ; voir par exemple le code source du programme de test fourni `ping-pong`.

1.2.3 Synchronisation au démarrage

Une difficulté au démarrage de l'application est de garantir qu'un processus n'envoie pas de message avant que ses voisins aient terminé le lancement de leurs écouteurs, afin d'être sûr que ses messages ne seront pas perdus. Pour simplifier, cette synchronisation (méthode `ConcurrentProcess.waitNeighbouring(String msg)`) est dans un premier temps réalisée manuellement (saisie au clavier d'un retour chariot par l'utilisateur, pour chaque processus, une fois tous les processus en attente ; voici son code à insérer dans votre classe `ConcurrentProcess` :

```
private final void waitNeighbouring(String msg){
    try {
      printOut(msg);
      new BufferedReader(new InputStreamReader(System.in)).
         readLine();
      trace("unblocked");
      ready.set(true);
    } catch (IOException e) {
    }
  }
```

Pour les tests, chaque processus sera lancé depuis un terminal distinct, puis une fois *tous* les processus lancés, on les débloquera (dans un ordre quelconque) en saisissant un retour chariot dans chaque terminal.

1.2.4 Communications

Les messages, modélisés par la classe `Message`, sont caractérisés par un numéro de processus expéditeur, un numéro de processus destinataire, un tag (chaîne de caractère) et un contenu (chaîne de caractères). Ils sont sérialisés avant écriture sur la socket (méthode `toBytes()`) et dé-sérialisés (méthode `fromBytes()`) à leur réception. Ils ne se perdent pas et ne se doublent pas sur un même canal (quand le canal est bien ouvert des deux côtés).

Le traitement applicatif à effectuer lors de la réception d'un message tagué [1] est décrit en implémentant l'interface `MessageHandler`. Ce traitement est mémorisé par le processus lors de son enregistrement (méthode `ConcurrentProcess.addMessageListener()`). Quand un écouteur reçoit un message, il appelle la méthode `ConcurrentProcess.receiveMessage()` qui déclenche le traitement associé au tag du message.

A noter que *pour un même tag, un seul traitement peut être exécuté à un moment donné* par un noeud, mais que *deux traitements associés à deux tags différents peuvent s'exécuter en parallèle*. L'envoi de message, non bloquant, est réalisé par la méthode `ConcurrentProcess.sendMessage()` et peut s'exécuter en parallèle de la réception de messages.

1.3 Programme de Test : ping-pong

Le package `ping_pong_V0` dont les sources vous sont intégralement fournis sur l'ENT contient un programme de test de votre blibliothèque simulant un ping-pong entre deux processus modélisés par la classe `PingPong`. Vous vous assurerez que l'ensemble des threads terminent proprement à l'issue de l'exécution. Aucun appel « sauvage » à la méthode `System.exit()` ne devra être utilisé où que ce soit dans le code.

Un script pour automatiser le lancement par `ssh` de vos programmes sur le cluster depuis une salle de TP vous est fourni. Il contient des variables à adapter aux conditions d'exécution.

1. Ceci correspond à la clause « sur réception de » des algorithmes vus en cours.

2 Deuxième partie : synchronisation automatique

Ne passez à cette partie que si la partie précédente est terminée et testée positivement.

Faîtes *une copie* de votre package `csp_V0` sous le nom `csp_V1` et modifiez cette nouvelle version de votre bibliothèque de communication d'après les indications suivantes.

2.1 Synchronisation d'un processus avec ses voisins

Dans la première partie du TP on a utilisé un événement externe (lecture d'un retour chariot) dans la méthode `ConcurrentProcess.waitNeighbouring()` pour s'assurer que tous les voisins d'un processus ont bien démarré leurs écouteurs avant de commencer à envoyer des messages. La seconde partie du TP consiste à remplacer cette synchronisation manuelle par un algorithme de synchronisation par échange de messages entre voisins.

Le principe est de réécrire cette méthode de telle manière que le processus informe ses voisins qu'il y est parvenu (donc qu'il a démarré ses propres écouteurs) et qu'il y soit bloqué tant que tout ses voisins ne l'ont pas informé à leur tour qu'ils sont prêts. De manière synthétique, voici le traitement à effectuer par chaque processus dans la méthode `waitNeighbouring()` :

- le processus commence par envoyer à chaque voisin, non déjà prêt, un message avec le tag prédéfini `SYNC_TAG` et le contenu `SYNC_MSG_READY`,
- pour les voisins dont il a été informé qu'ils sont prêts, c-a-d. dont il a reçu (voir méthode `SyncMessageHandler.onMessage()` ci dessous) de leur part un message `( SYNC_TAG, SYNC_MSG_READY)`, il leur renvoie un message `(SYNC_TAG,SYNC_MSG_ACK)`,
- ensuite le processus reste bloqué tant qu'il reste un voisin non prêt.

Les messages de synchronisation reçus par un processus nécessitant un traitement particulier, on ajoute un handler de message de la nouvelle classe `csp_V1.SyncMessageHanler` à la table des handlers du processus à sa création. La méthode `onMessage()` de ce handler est appelée sur réception d'un message tagué par `SYNC_TAG` ; dans cette méthode on traite les messages de synchronisation `SYNC_MSG_READY` et `SYNC_MSG_ACK`, en y répondant, si nécessaire, et en décrémentant le compteur du nombre de voisins non prêts quand on détecte qu'un nouveau voisin est prêt.

Le protocole décrit ci dessous nécessite au plus deux échanges de messages entre chaque processus pour qu'ils se synchronisent. Les diagrammes temporels représentant tous les cas possibles d'ordonnancement des envois et réceptions de messages de synchronisation entre voisins sont représentés sur la figure 2. Pour vous guider dans la programmation, l'ensemble des cas possibles et leur description est modélisé par le type énuméré `csp_V1.SyncState` (fourni sur l'ENT) ; l'état courant de la situation avec chaque voisin est mémorisé dans la table `ConcurrentProcess.sync_state_table`. A noter qu'un processus attend toujours d'avoir reçu un ACK d'un voisin avant de considérer qu'il est prêt.

Avant de vous lancer dans la programmation, représentez sur feuille un automate d'états finis modélisant l'état d'un processus en attente de synchronisation. Cet automate vous sera demandé en TP.

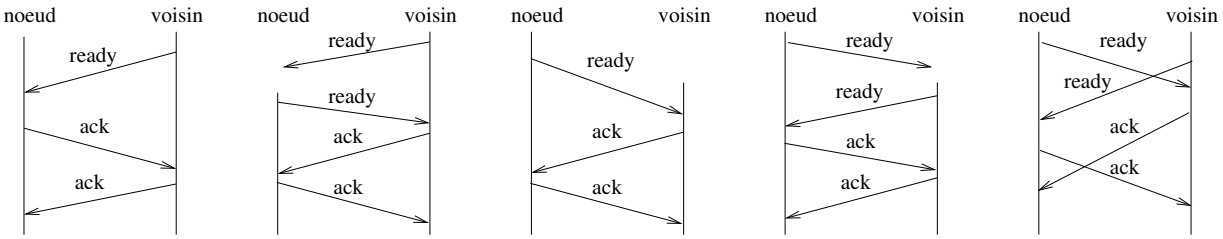


Fig. 2: Diagrammes temporels des synchronisations au démarrage

2.2 Tests : ping-pong et construction d'un réseau

Le programme de test « PingPong » utilisé dans la première partie du TP peut servir de premier test. Vous trouverez sur l'ENT une nouvelle version `ping_pong_V1.zip` où l'importation du package `csp_V0` dans les sources a été remplacée par celle du package `csp_V1` ; le reste est identique. Vous devrez modifier vous même les scripts de test fournis avec `ping_pong_V0`.

Par ailleurs vous testerez votre package `csp_V1` avec le programme `BuildNet` fourni sur l'ENT ; ce programme construit le réseau de la figure 3, affiche le nombre de messages émis et reçus par chaque processus avant synchronisation, puis effectue un échange avec accusé de réception avec chaque voisin.

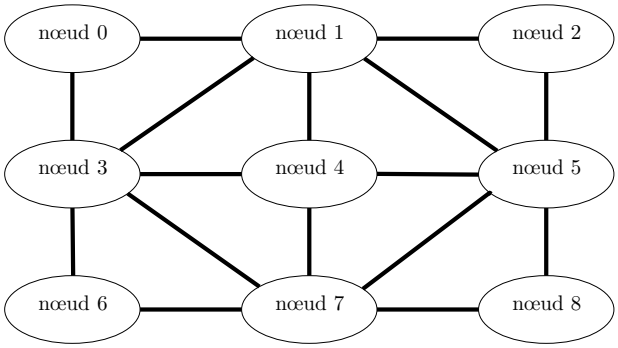


Fig. 3: Réseau utilisé pour le test de synchronisation