# TP3
# Consensus et détection de fautes

Dans ce TP, nous allons implémenter l'algorithme du coordinateur tournant vu en cours, à l'aide d'un détecteur de fautes rudimentaire.

Commencez par créer un nouveau projet, par exemple par copié-collé d'un TP précédent.

## 1 Simulation de pannes

Nous allons commencer par créer une classe de noeuds très simples, qui peuvent tomber en panne. Pour simuler les pannes, nous allons équiper chaque noeud d'un attribut booléen :

```
private boolean alive;
```

Lorsque cet attribut `alive` vaut `true`, le noeud est en vie, et s'il vaut `false` le noeud est en panne. `alive` doit être initialisé à `true`.

Lorsque l'utilisateur appelle la fonction `onSelection()` à l'aide d'un ctrl+clic, `alive` passe à `false` et le noeud change de couleur pour devenir rouge.

Dans le reste du TP, il faudra s'assurer que les noeuds en panne ne font plus rien. Pour cela, vous pouvez simplement conditionner le contenu de vos fonctions `onMessage()` et `onClock()` à la valeur de `alive`.

```
public void onMessage(Message m) {
    if (alive) {
        ...
    }
}

public void onClock() {
    if (alive) {
        ...
    }
}
```

**Question 1.** Programmez une classe de noeuds capables de «tomber en panne» suivant les instructions ci-dessus.

**Question 2.** Créez un programme main qui instancie 5 noeuds et les connecte tous avec des liens bidirectionnels (graphe complet), puis testez votre programme. Vérifiez que les noeuds deviennent rouge lors d'un ctrl+clic.

## 2 Détecteur de fautes Heartbeat

Maintenant que nos noeuds peuvent tomber en panne, il va falloir faire en sorte de détecter ces pannes. Pour cela, nous allons utiliser la méthode Heartbeat. Chaque noeud envoie régulièrement un message (un «battement de coeur») à tous les autres pour leur signaler qu'il est en vie. Chaque fois qu'un noeud A n'a pas reçu de nouvelles d'un noeud B depuis trop longtemps, il suspecte B d'être en panne. Si A reçoit par la suite un message de B, il corrige alors son erreur et fait à nouveau confiance à B.

Cette méthode garantit que les noeuds en panne seront détectés, puisqu'ils vont cesser d'émettre des Hearbeat.

Chaque noeud devra disposer d'un ensemble `trusted`, contenant tous les noeuds à qui il fait actuellement confiance.

```
private Set<Node> trusted;

public void onStart() {
    trusted = new HashSet<>(); // N'oubliez pas d'initialiser l'ensemble!
    ...
}
```

Pour simuler le temps, JBotSim dispose de sa propre horloge. À chaque tick de cette horloge, la méthode `onClock()` de chaque noeud est appelée. Vous pouvez donc placer dans `onClock()` le code que vous voulez exécuter à intervalles réguliers.

Pour notre détecteur, chaque noeud doit envoyer un Heartbeat tous les 10 instants. Tous les 20 instants, chaque noeud met à jour son ensemble `trusted` pour y inclure seulement les processus dont il a reçu un message dans les 20 derniers instants.

Afin de compter le temps, vous pouvez utiliser un attribut entier comme compteur : si `onClock()` a été appelé 10 fois, c'est que 10 instants se sont écoulés.

Il vous est recommandé d'utiliser un deuxième ensemble pour garder la trace des messages reçus dans les 20 derniers instants.

Pour vérifier le fonctionnement du détecteur, pensez à faire des affichages (par exemple, tous les 20 instants lorsque `trusted` change). Vous pouvez aussi utiliser `setLabel()` qui permet de changer le contenu de l'infobulle d'un noeud dans le canevas JBotSim.

**Question 1.** Programmez un détecteur Heartbeat et testez-le en simulant une panne lors de l'exécution.

Par défaut, JBotSim est configuré pour fonctionner en mode synchrone : tous les messages envoyés mettent exactement 1 instant à arriver à destination.

**Question 2.** Dans ces conditions, notre détecteur Heartbeat peut-il donner lieu à des fausses suspicions ?

Afin de rendre les choses plus intéressantes, nous allons dire à JBotSim de passer en mode asynchrone. Ajoutez les instructions suivantes dans votre main, avant le `tp.start()` :

```
MessageEngine me = new AsyncMessageEngine(tp, 5, AsyncMessageEngine.Type.FIFO);
tp.setMessageEngine(me);
```

**Question 3.** Testez à nouveau votre programme. Que constatez-vous ?

# 3 Algorithme du coordinateur tournant

**Pour les étudiants qui travaillent avant la séance de TP** : cet exercice est long et difficile, n'essayez pas de tout faire chez vous à moins d'être très à l'aise. On vous aidera en TP.

Dans cet exercice, nous allons utiliser le détecteur programmé précédemment pour implémenter l'algorithme du coordinateur tournant. Il s'agit d'un algorithme de consensus présenté dans le CM4.

Référez vous à la description détaillée de l'algorithme, en **pages 16-17 du résumé de cours**. Je vous recommande de relire ce passage du cours et de vous assurer de bien comprendre l'algo avant de continuer l'exercice.

L'algorithme du coordinateur tournant s'appuie sur deux hypothèses :

- une majorité de processus doivent être corrects (dans notre cas, il suffit de ne pas faire tomber trop de processus en panne) ;
- un détecteur de fautes $\diamond S$.

Le détecteur de fautes que nous avons implémenté à l'exercice précédent est **plus faible** que $\diamond S$. Il est néanmoins suffisant pour faire fonctionner un coordinateur tournant sans garantie de terminaison. Il se peut que les processus ne parviennent pas à une décision en un temps fini, mais si ils décident, alors leur décision sera cohérente.

## 3.1 Héritage

Afin de bénéficier de notre détecteur de fautes, nous allons programmer le coordinateur tournant dans une classe de noeud qui héritera de celle de l'exercice précédent.

```java
public class NoeudCoordinateur extends NoeudHeartbeat {
    ...
    public void onStart() {
        super.onStart(); // Cette ligne est nécessaire pour que le onStart() de la classe parente
            ↪soit exécuté.
        ...
    }

    public void onClock() {
        if (alive) { // Ne pas oublier que les noeuds morts ne font rien.
            super.onClock();  // Pour que le onClock() de la classe parente soit exécuté.
            ...
        }
    }

    public void onMessage(Message m) {
        if (alive) {
            super.onMessage(m); // Idem.
            ...
        }
    }
    ...
}
```

Notre algorithme aura besoin des attributs `alive` et `trusted` de la classe parente. Pour pouvoir y accéder, il faut modifier leur accessibilité de `private` en `protected`.

```java
protected boolean alive;
protected Set<Node> trusted;
```

Les autres attributs de la classe parente ne seront pas utilisés dans la nouvelle classe, et peuvent donc rester `private`.

**Question 1.** Remplacez NoeudHeartbeat (ou le nom de votre classe parente) par NoeudCoordinateur dans votre main, puis vérifiez que tout fonctionne toujours.

## 3.2 Attributs et changements de ronde

La description de l'algorithme dans le cours indique que chaque processus doit maintenir deux variables locales :
- $v$ : la valeur que ce noeud essaie actuellement de décider ;
- $ts$ (timestamp) : le numéro de ronde durant lequel le noeud a adopté la valeur $v$.

Dans une implémentation concrète de l'algorithme, nous aurons besoin de quelques variables additionelles :
- *ronde* : le numéro de la ronde asynchrone actuellement exécutée par ce noeud ;
- *phase* : la phase dans laquelle se trouve le noeud (estimation, proposition, ack, décision) ;

Pour simuler un input impossible à prédire, nous allons initialiser $v$ à une valeur aléatoire.

```java
// Ma valeur proposée est un nombre au hasard entre 0 et 100.
v = (int) Math.round(Math.random() * 100);
```

**Question 2.** Déclarez ces attributs, et initialisez les.

Afin de faciliter la programmation du reste de l'algorithme, nous allons programmer une méthode `nouvelleRonde()` qui sera appelée lorsqu'un processus change de ronde. Pour le moment, cette fonction se contentera d'incrémenter `ronde` et de produire un affichage (par exemple : «Le noeud 3 passe à la ronde 5»).

**Question 3.** Programmez cette fonction, puis appelez la à la fin du onStart().

À chaque ronde, le coordinateur est le processus dont l'id est égal au numéro de ronde modulo le nombre de processus. Pour vous faciliter la vie, vous pouvez utiliser la fonction suivante :

```java
// Petite fonction utilitaire qui retourne le coordinateur en fonction de la ronde.
private Node coordinateur(int ronde) {
    // Hypothèse: les noeuds connaissent le nombre de noeuds.
    int id = ronde % getTopology().getNodes().size();
```

```
    return getTopology().findNodeById(id);
}
```

Un noeud qui ne fait plus confiance au coordinateur (c'est à dire que le coordinateur n'est pas dans son `trusted`) doit changer de ronde, pour ne pas être bloqué par un coordinateur en panne.

**Question 4.** Modifiez la fonction onClock() pour qu'un noeud qui suspecte son coordinateur, change de ronde.

### 3.3 Phases et messages

En tout, votre algorithme devra traiter 5 types de messages différents : les messages Heartbeat (déjà traités dans l'exercice précédent), et un type de message par phase (estimation, proposition, ack et décision).

N'oubliez pas de différencier les messages Heartbeat, qui sont traités par `super.onMessage(m)`, des autres types de messages.

Un message d'estimation contiendra une valeur, le numéro de la ronde à laquelle cette valeur a été proposée, et le numéro de la ronde à laquelle ce message d'estimation est envoyé.

Un message de proposition contient une valeur et un numéro de ronde.

Un message d'ack contient un numéro de ronde, et un booléen indiquant si la proposition est acceptée ou non.

Un message de décision contient simplement une valeur.

Vous pouvez gérer ces types de messages dans votre programme comme vous le souhaitez. Par exemple, vous pouvez faire une classe de messages pour chaque type, ou bien une seule classe capable de contenir tous les paramètres possibles.

Votre méthode `onMessage()` devra être structurée comme suit :

```
public void onMessage(Message m) {
    if (alive) {
        if (m est un message Heartbeat)
            super.onMessage(m);
        else if (m est un message estimation) {
            ...
        } else if (message proposition) {
            ...
        } else if (message ack) {
            ...
        } else if (message décision) {
            ...
        }
    }
}
```

### 3.4 Phase d'estimation

Chaque ronde commence par une phase d'estimation. Dans cette phase, tous les processus envoient leur $v$, $ts$ et *ronde* au coordinateur, qui devra alors choisir l'une de ces valeurs.

**Question 5.** Complétez votre fonction nouvelleRonde() pour qu'un noeud qui change de ronde mette à jour son attribut phase (estimation) et envoie un message au coordinateur.

Le coordinateur attend de recevoir un message d'estimation d'une majorité de processus (en n'oubliant pas de compter son propre vote) avant de continuer.

Attention, les rondes sont asynchrones. Il se peut qu'un processus soit à la ronde 4, pendant qu'un autre est à la ronde 27. Il faut donc bien s'assurer de ne traiter que les messages d'estimation correspondant à la ronde actuelle.

Si je reçois un message d'une ronde passée, je peux l'ignorer. En revanche, si je reçois un message d'une ronde future (envoyé par un processus en avance sur moi), alors il faut stocker ce message pour plus tard. Je ne peux pas traiter ce message tout de suite car je ne suis pas encore à la bonne ronde, mais je ne peux pas non plus l'ignorer au risque de manquer de messages d'estimation quand j'arriverai à cette future ronde.

Pour résoudre ce problème, nous allons avoir besoin de stocker tous les messages d'estimation reçus, en les rangeant par ronde. Ajoutez un nouvel attribut :

```java
private Map<Integer,Set<Message>> estimationsReceived;

public void onStart() {
    ...
    estimationsReceived = new HashMap<>(); // Penser à initialiser.
    ...
}
```

En Java, une Map est une table de hachage qui associe des clés à des valeurs. Ici, les clés sont des entiers (les numéros de ronde), et les valeurs sont des ensemble de message (l'ensemble des messages reçus pour cette ronde).

Exemples de manipulations :

```java
estimationsReceived.get(3); // Retourne l'ensemble des messages reçus pour la ronde 3.

// Attention, chaque ensemble doit être initialisé avant d'être utilisé.
if (estimationsReceived.get(4) == null)
    estimationsReceived.put(4, new HashSet<>());

estimationsReceived.get(4).add(m); // J'ajoute un nouveau message correspondant à la ronde 4.
int nombre = estimationsReceived.get(4).size(); // Je compte les messages reçus pour la ronde 4.
```

Lorsqu'un noeud reçoit un message d'estimation, il faut donc suivre les étapes suivantes :

1. s'il s'agit du premier message que je reçois pour cette ronde, j'initialise l'ensemble correspondant ;
2. j'ajoute le nouveau message au messages reçus pour cette ronde ;
3. je vérifie si j'ai reçu un message d'une majorité de processus pour ma ronde actuelle, sans oublier de me compter moi-même.
4. Si j'ai reçu une majorité, je parcours les messages reçus pour trouver la valeur qui a le timestamp le plus élevé. Je mets à jour mon $v$ et mon $ts$ avec cette nouvelle valeur. J'annonce dans la console que j'ai reçu une majorité d'estimations.

Faites bien la différence entre la ronde du message reçu, et la ronde dans laquelle le noeud se trouve.

**Question 6.** Complétez votre méthode onMessage() pour traiter les messages d'estimation reçus.

## 3.5 Phase de proposition

Lorsqu'un coordinateur a reçu une majorité de messages d'estimation, il peut passer en phase de proposition. Dans cette phase, il envoie à tous les processus (`sendAll()`) un message contenant sa valeur et son numéro de ronde.

Lorsqu'un noeud reçoit un message de proposition du coordinateur, il doit vérifier le numéro de ronde de cette proposition.

Si la proposition correspond à ma ronde actuelle, il me faut alors mettre à jour mon $v$ et mon $ts$ pour adopter la valeur et le numéro de ronde contenus dans le message. J'envoie ensuite un message ack au coordinateur contenant mon numéro de ronde, et un booléen à `true` pour indiquer que j'ai accepté cette proposition. Finalement, je peux passer à la ronde suivante avec `nouvelleRonde()` : le coordinateur n'a plus besoin de moi pour cette ronde.

Si la proposition reçue correspond à une ronde passée, je dois alors répondre par la négative. Je réponds un message ack à l'émetteur de la proposition, contenant le numéro de ronde de la proposition reçue, et un booléen à `false`.

Si la proposition reçue correspond à une ronde future, il faut la stocker pour plus tard. Pour cela nous allons ajouter un nouvel attribut :

```java
private Map<Integer, Message> propsReceived;

public void onStart() {
    ...
    propsReceived = new HashMap<>(); // Penser à initialiser.
    ...
}
```

Le stockage est plus simple cette fois ci. Pas besoin de `Set`, puisque chaque noeud ne doit recevoir qu'un seul message de proposition par ronde. Les messages peuvent alors simplement être stockés comme suit :

```
propsReceived.put(3, m); // Je stocke le message reçu pour la ronde 3.
propsReceived.get(3); // Je récupère le message reçu pour la ronde 3.
```

**Question 7.** Complétez votre méthode onMessage() pour traiter les messages de proposition reçus.

Maintenant que les messages de proposition des rondes futures sont correctement stockés, il faut les traiter au moment du changement de ronde.

Lorsqu'un noeud change de ronde, il doit vérifier s'il a déjà reçu un message de proposition pour cette ronde. S'il c'est le cas, il ne passe pas par la phase d'estimation : il va immédiatement traiter la proposition. Dans le cas où le coordinateur est suspecté, il faut répondre par la négative. Sinon, il faut répondre un ack positif et mettre à jour les valeurs $v$ et $ts$. Inspirez vous de ce que vous avez fait dans `onMessage()`.

Dans les deux cas, on peut ensuite passer directement à la ronde suivante par un appel récursif à `nouvelleRonde()`.

**Question 8.** Modifiez votre méthode nouvelleRonde() pour traiter le cas où un message de proposition a déjà été reçu pour la nouvelle ronde.

## 3.6 Phase d'acks

Le coordinateur attend de recevoir des messages acks d'une majorité de processus. Les acks reçus correspondent toujours à une ronde passée ou actuelle : il est impossible de recevoir un ack pour une ronde dans laquelle je n'ai pas encore envoyé de proposition. Le stockage des acks reçus est donc très simple :

```
private Set<Message> acksReceived;

public void onStart() {
    ...
    acksReceived = new HashSet<>(); // Penser à initialiser.
    ...
}
```

Lors de la réception d'un ack, on vérifie le numéro de ronde. S'il s'agit d'une ronde passée, le message est ignoré. S'il s'agit de la ronde actuelle, le message est stocké dans `acksReceived`. Il faut ensuite vérifier si une majorité de acks a été reçue, sans oublier de se compter soi-même.

Si une majorité de réponses a été reçues, il faut l'annoncer par un affichage console, puis vérifier que toutes ces réponses sont positives. Si oui, c'est gagné ! On passe alors en phase de décision. S'il y a des réponses négatives, c'est un échec : on passe alors à la ronde suivante avec `nouvelleRonde()`.

**Attention :** l'ensemble `acksReceived` doit être vidé lors de l'envoi d'une nouvelle proposition, sans quoi les acks des rondes précédentes seront réutilisés.

**Question 9.** Complétez votre méthode onMessage() pour traiter les messages acks reçus.

## 3.7 Phase de décision

Lorsque le coordinateur reçoit une majorité de acks positifs, il passe en phase de décision. Il peut donc décider sa valeur $v$ en l'annonçant dans la console, puis diffuser à tous un message de décision contenant cette valeur.

Lorsqu'un processus reçoit un message de décision, quoi qu'il arrive, il décide la valeur contenue dans le message en l'annonçant dans la console.

Un processus qui a décidé ne participe plus à l'algorithme. Pour implémenter cela simplement vous pouvez par exemple passer `alive` à `false` lorsque le processus décide.

**Question 10.** Terminez votre algorithme en complétant onMessage() pour traiter les messages de décision. Testez votre code et vérifiez que les noeuds parviennent à décider une valeur.