Réalisation d’un intergiciel (middleware)

Dans la prise en main du bus asynchrone, vous avez certainement implémenté l’ensemble des méthodes d’émissions et réceptions dans la classe Process. Hors ceci est une tâche du middleware et non du processus.

Repartons donc sur de bonnes bases et enrichissons ce que nous (vous) avons fait précédemment. Créez une classe « Com » qui sera votre communicateur ; soit la classe qui contient l’ensemble des méthodes de communication. Ainsi chaque processus pourra instancier un communicateur pour l’envoi et la réception de message. Votre classe « Process » devra impémenter l’interface Lamport qui définit les méthodes « get_clock() », « set_clock() », « lock_clock() » et « unlock_clock() » ; les méthodes « lock_clock() » et « unlock_clock() » gèrent un sémaphore pour accéder à la ressource critique locale qu’est l’horloge de Lamport. Cette gestion se fera au sein du communicateur en utilisant les méthodes de votre processus, afin que le communicateur mette à jour l’horloge du processus.

Dans notre middleware, nous souhaitons que le communicateur fournisse, à son processus, une boite aux lettres (B.a.L.) pour tous les messages asynchrones. Ainsi le processus pourra piocher à sa guise les messages qu’il a reçus dans cette boite.

Concernant les messages, il est conseillé d’avoir une classe abstraite de messages génériques. Cette dernière contiendra toutes les informations communes à tout message (Payload, stamp…). Tous les messages supportés par votre communicateur hériteront de cette classe abstraite.

Le communicateur doit fournir aux processus les méthodes de communication asynchrone :

- broadcast(Object o) : qui envoie l’objet « o » dans toutes les B.a.L. des autres processus.
- sendTo(Object o, int dest) : qui envoie l’objet « o » dans la B.a.L. du processus « dest ».

Il doit fournir aux processus les services de :

- section critique distribuée avec les méthodes « requestSC() » et « releaseSC() ». La méthode « requestSC() » doit bloquer le processus jusqu’à obtention de la section critique. L’utilisation d’un jeton sur anneau au sein du communicateur est la méthode la plus simple pour répondre à ce problème. Cependant, la gestion du jeton doit être gérée par un thread tiers et les messages contenant le jeton ne doivent pas impacter l’horloge du processus, car ce sont des messages systèmes,
- synchronisation avec la méthode « synchronize() » qui attend que tous les processus aient invoqué cette méthode pour tous les débloquer.

Le communicateur doit fournir aux processus les méthodes de communication synchrone (donc bloquantes):

- broadcastSync(Object o, int from) : qui si le processus a l’identifiant « from », il attend que tous les autres invoquent cette méthode et envoient l’objet « o » à tous les autres processus ; si le processus n’a pas l’identifiant « from » il attend de recevoir le message de « from ».

- sendToSync(Object o, int dest) et recevFromSync(Object o, int from) : qui respectivement envoie le message à « dest » et bloque jusqu'à ce que « dest », reçoit le message et bloque jusqu'à ce que « from » envoie le message.

Le communicateur doit fournir un système de numérotation automatique et consécutive, quand un processus se connecte, il reçoit un numéro unique. La numérotation commençant à 0 pour le premier processus. Ce système sera couplé à un « heartbit », c'est à dire un message système envoyé périodiquement par les processus à tous les autres pour prouver qu'ils sont encore vivants. Si un des processus ne communique plus, la numérotation de ceux encore vivants doit être corrigée.

Pour toutes les fonctions que vous avez réalisées, vous rendrez le code Java (ou autre) ainsi que la JavaDoc associée à ces fonctions. Un exemple d'utilisation de vos fonctions, comme le jeu de dé proposé dans la prise en main de Guava sera le bienvenu.