Programmation répartie

HEIG-VD / PRR / Labo 1

Temps à disposition : 10 périodes.
Travail à réaliser par groupe de 2 étudiants.
La présence aux labos est obligatoire. En cas d'absences répétées, l'étudiant sera pénalisé.
En cas de copie manifeste entre les rendus de deux labos, tous les étudiants des deux groupes se verront affecter la note de 1.
Distribué le mercredi 18 septembre 2019 à 14h55.
A rendre le mercredi 16 octobre 2019 à 16h30. *Une démonstration pourra être demandée.*

Forme du rendu : impression des fichiers sources à remettre en séance, archive du projet par email à l'assistant et à l'enseignant. Readme indiquant comment l'utiliser et précisant ce qui a été réalisé, ce qui fonctionne et ce qui reste à faire.

Objectifs

- Écrire un premier programme de complexité moyenne en Go (Golang) et se familiariser avec un environnement de programmation Go
- Réaliser ses premières communications par UDP et par diffusion partielle (multicast).

Critères d'évaluation

La note du labo sera constituée de 2 notes au demi-point prêt, chacune étant comprise entre 0 et 2,5 points : une pour le bon fonctionnement du programme et une pour la qualité du rendu. La note du labo (sur 6 points) sera obtenue par : 1 + la somme de ces 2 notes.

Le bon fonctionnement du programme est obtenu s'il couvre toutes les fonctionnalités de l'énoncé du labo, s'exécute de façon performante et ne présente pas de bugs.

La qualité du rendu sera évaluée en tenant compte des points suivants :

- Facilité et rapidité de la mise en œuvre du programme (activation, aide, paramétrage), en particulier si on utilise un seul PC pour le tester (options par défaut bien choisies).
- Facilité et rapidité de la vérification de son fonctionnement : traces éventuellement paramétrables, datées et signées, mais en tout cas claires et adéquates.
- Possibilités de paramétrage pour simuler des conditions réseau réelles (délais de transmission, dérive d'horloge, pannes, …) dans les limites de l'énoncé.
- Réalisation de tests automatisés avec simulation de site ou d'une partie de l'application (mocking).
- Conception du code source (structure et décomposition). Possibilité de réutilisation d'une partie du code avec un autre énoncé (autre couche de transport réseau, …). Cependant, il ne doit pas faire plus que ce qui est demandé, ni être prévu pour des extensions hypothétiques.
- Qualité, simplicité, concision et lisibilité du code source. Conformité au format de code source et aux bonnes pratiques préconisées pour le langage.
- Documentation des entêtes de modules et de fonctions. Commentaires adéquats des instructions : un commentaire trivial ou inutile est aussi pénalisant que l'absence d'un commentaire nécessaire à la compréhension.
- Lisibilité du code imprimé : pagination adéquate (fonction sur une seule page ou bien découpée), entêtes et fin de pages, titres des modules et fonctions apparents, police adéquate et indentation bien choisie (pas de retour ligne automatique).

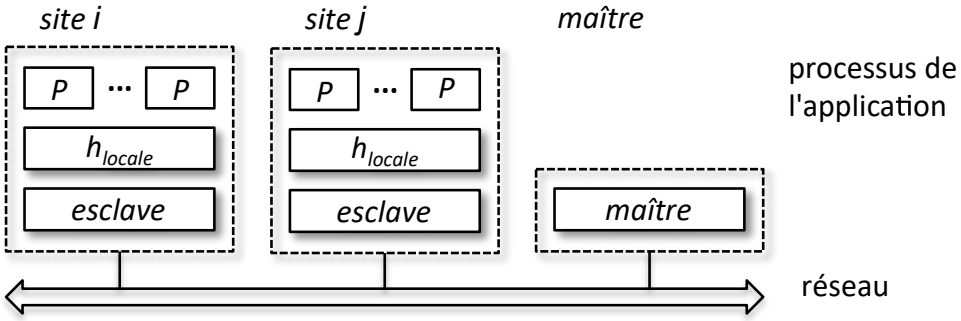
Énoncé du problème

Nous souhaitons implémenter un algorithme simple permettant de synchroniser approximativement les horloges locales des tâches d'une application répartie. Comme nous le savons, chaque site d'un environnement réparti possède sa propre horloge système, mais aussi, cette horloge a un décalage et une dérive qui lui est propre. Le but de notre algorithme est de rattraper ce décalage sans pour autant corriger l'horloge du système. Pour ce faire, nous distinguons 2 horloges. L'horloge système h_{sys} est l'heure retournée par un appel système : cette horloge désigne la minuterie mise à jour par le système d'exploitation d'un site. Sous un système protégé, il faut avoir les privilèges administrateurs pour le modifier et, pour contourner ce problème, une tâche applicative peut interpréter le temps comme la valeur de l'horloge système sur le site où elle réside additionnée à un décalage. Dans ce qui suit, le résultat de cette opération est appelé l'horloge locale. Ainsi pour la tâche applicative *i*, nous avons : $h_{locale}(i) = h_{sys}(\text{site de } i) + décalage(i)$

La synchronisation des horloges revient alors à trouver *décalage*(*i*) pour chaque tâche *i* de telle sorte que $h_{locale}(i)$ est identique pour toutes les tâches formant l'application répartie.

Protocole de synchronisation

Les sites se divisent en 2 groupes : le maître qui est unique et les esclaves qui sont subordonnés au maître pour l'exécution de l'algorithme. Cette structure se représente ainsi :



Si *P* désigne les tâches de l'application, chaque tâche peut alors interroger son gestionnaire d'horloge (*esclave* sur la figure) résidant sur le même site qu'elle. Elle obtient alors une heure qui est approximativement synchrone avec les autres tâches sur d'autres sites de l'application. Ce sont alors uniquement les esclaves et le maître qui doivent se synchroniser.

La procédure de synchronisation est réalisée en 2 étapes. La première détermine l'écart temporel, *écart*(*i*), entre le maître et les esclaves, alors que la seconde corrige le délai de transmission et les latences, *délai*(*i*), entre le maître et ses esclaves. Leur somme donne $décalage(i) = écart(i) + délai(i)$, pour le site *i*.

Périodiquement, le maître diffuse 2 messages. Le premier message est de type SYNC et contient un identifiant. Le second message, FOLLOW_UP, diffusé immédiatement après le premier, contient l'heure, $t_{maître}$, auquel le maître a émis le message SYNC.

```
boucle
    identifiant ← identifiant + 1
    msg ← <SYNC,identifiant>
    t_maître ← h_sys(maître)
    diffusion de msg
    msg ← <FOLLOW_UP,t_maître,identifiant>
    diffusion de msg
    attente de k secondes
fin boucle
```

La correction de l'écart se réalise indépendamment par tous les esclaves lors de la réception du message FOLLOW_UP. Les esclaves obtiennent $t_i = h_{sys}(i)$ quand ils reçoivent le message SYNC, puis à la réception du message FOLLOW_UP, ces esclaves calculent

$$écart(i) = t_{maître} - t_i.$$

Sur un environnement idéal (temps de calcul instantané et délai de transmission nul), les horloges du maître et de ses esclaves seraient synchrones à l'issue de cette première étape.

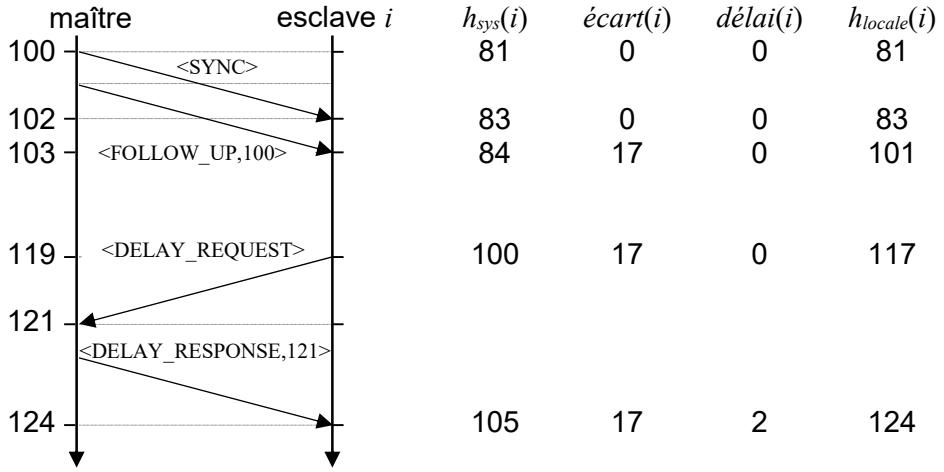
La seconde étape est lancée par les esclaves pour mesurer approximativement la latence de transmission entre les tâches esclaves et le maître. Un message de type DELAY_REQUEST est émis au maître. À la réception de ce message, le maître renvoie un message de type DELAY_RESPONSE à l'esclave contenant l'heure qu'il a reçu le message DELAY_REQUEST. L'esclave peut ensuite raffiner son décalage par rapport au maître : si t_{es} est l'heure locale d'émission de DELAY_REQUEST et t_m celle de sa réception par le maître, alors

$$délai(i) = (t_m - t_{es}) / 2.$$

Notons que l'esclave insère aussi un identifiant dans le message DELAY_REQUEST. Cet identifiant est renvoyé par le maître et sert de contrôle.

Cette seconde étape est exécutée irrégulièrement et à des intervalles de temps supérieurs à k. L'esclave entame la seconde étape pour la première fois après la première étape et après un temps aléatoire tiré de l'intervalle $[4k,60k]$. Toutes les fois subséquentes, cette étape se fait après la précédente et après un temps aléatoire tiré depuis le même intervalle.

La figure ci-dessous illustre un échange typique.



Remarques sur le protocole

- En réalisant des diffusions, la variabilité du temps de transmission entre des sites différents est réduite, si cette diffusion se fait sur un média de diffusion commun.
- Le protocole suppose que les latences sont symétriques, c.-à-d. que les délais des communications entre le maître et ses esclaves ne dépendent pas du sens de la communication.
- Le protocole décrit correspond à la norme IEEE 1588 admise en 2002 utilisé pour synchroniser des horloges temps réel sur un bus de terrain et réaliser la cohérence temporelle des acquisitions. Ce protocole est aussi connu sous le nom de « Precision Time Protocol » ou PTP.

Travail à faire

Implémenter cet algorithme en Go avec les hypothèses suivantes :

1. Le maître et ses esclaves seront prévus pour s'exécuter sur des sites différents reliés par un LAN permettant la diffusion. Le nombre d'esclaves peut varier.
2. Le maître est lancé en premier. S'il est en panne, les esclaves ne sont plus synchronisés mais le

seront de nouveau lorsque le maître sera relancé. Si un esclave est en panne, cela ne doit pas perturber le fonctionnement du maître.

3. Les communications du maître vers ses esclaves sont des diffusions partielles (multicast UDP).
4. Les messages DELAY_REQUEST et DELAY_RESPONSE se font par datagrammes point-à-point UDP.
5. Les messages doivent être les plus courts possibles. Les chaînes de caractères ne doivent pas se substituer à des nombres ! Il vous faudra travailler avec des octets.

Note pour le point 1 : La mise en œuvre d'un réseau LAN permettant la diffusion n'étant pas facile à mettre en place, le bon fonctionnement de votre programme sera vérifié en local sur un seul PC (tous les sites paramétrés à l'adresse localhost).