SCIENCES SORBONNE UNIVERSITÉ

Projet ARA 2019–2020

Jonathan Lejeune, Arnaud Favier

Objectifs

L'objectif principal de ce projet est d'étudier les performances de différents algorithmes d'élection de leader sur des réseaux mobiles ad hoc, appelés plus couramment MANET (Mobile Ad hoc NETworks).

Consignes générales

- Ce travail est à faire seul ou (de préférence) en binôme. Les trinômes ne sont pas autorisés.
- Les livrables sont à rendre sur Moodle pour le 26 janvier 2020, à 23h59 au plus tard
- Il est attendu de rendre 2 fichiers :
  - ◇ Une archive des **sources java** (éviter les archives trop volumineuses, en évitant par exemple d'inclure les .class), accompagnées d'un fichier texte "Readme" indiquant comment compiler le projet et lancer les différentes simulations (votre projet doit pouvoir se compiler/lancer en dehors d'Eclipse) ;
  - ◇ Votre rapport, au format pdf, concis, dans lequel vous devez répondre aux questions posées dans le sujet.

Introduction

Un réseau ad hoc est composé de nœuds possédant chacun un émetteur radio sans fil. Les communications ne se basent sur aucune infrastructure fixe et il n'y a aucune entité centrale responsable du routage des messages. En effet, les nœuds peuvent communiquer directement avec leurs voisins, c'est-à-dire les nœuds présents dans leur rayon d'émission. Deux nœuds non voisins peuvent donc communiquer seulement si d'autres nœuds situés entre eux peuvent relayer les messages de hop en hop. De plus, les nœuds ont des contraintes énergétiques, car ils se basent sur une batterie qui peut se décharger plus ou moins vite si l'émetteur radio est trop souvent sollicité. Dans les MANET, les nœuds sont mobiles. Les liens réseau entre voisins sont dynamiques et peuvent donc apparaître ou disparaître au cours du temps en fonction du mouvement de chaque nœud. Aucune garantie ne peut donc être assurée sur une connectivité de bout en bout.

L'élection de leader est un problème fondamental des systèmes distribués. Il permet à un groupe de nœuds de désigner un identifiant parmi eux qui prend le rôle de leader. Un leader peut être utile par exemple pour implémenter un algorithme de consensus (Paxos), désigner un coordinateur, centraliser des informations sur le groupe ou bien faire office de représentant du groupe pour des entités externes. Dans un MANET, le caractère dynamique des nœuds est très important, ce qui implique que l'algorithme d'élection soit très tolérant aux changements de topologie, en assurant en permanence qu'à terme un groupe connexe de nœuds reconnaissent le même leader.

Préparation du projet

Créez un nouveau projet Eclipse, configurez le build path pour prendre en compte les jar de PeerSim (voir TP de prise en main) et y copiez les sources associées au projet. Assurez-vous que la compilation se passe bien (absence de croix rouges). Dans les sources vous trouverez les packages suivants :

- `ara.manet` : package contenant des classes ou interfaces utilitaires :
  - ◇ la classe `GraphicalMonitor` : control peersim permettant d'avoir un aperçu graphique de l'état du système. Ceci peut vous servir pour les phases de debug car il permet de mettre en pause, accélérer, ralentir la simulation.
  - ◇ la classe `MANETGraph` : permet de représenter si besoin le système sous forme de graph PeerSim permettant ensuite d'appliquer des algorithmes de graphes (par exemple : la connexité). Vous pouvez consulter l'interface `peersim.graph.Graph` et la classe `peersim.graph.GraphAlgorithms` pour plus d'informations.
  - ◇ l'interface `Monitorable` : utilisée par le contrôleur graphique, elle permet de récupérer des informations sur l'état du nœud notamment sur la partie applicative et de l'afficher sur la fenêtre graphique. Chaque état applicatif est associé à une couleur sur le moniteur graphique.
- `ara.manet.communication` composé de l'interface `Emitter` qui est un protocole permettant de simuler l'émission de messages dans un rayon géographique donné.
- `ara.manet.detection` composé de deux interfaces :
  - ◇ `NeighborProtocol` : représente un protocole permettant de détecter les voisins directs d'un nœud
  - ◇ `NeighborhoodListener` : représente un protocole (applicatif) sensible aux changements du voisinage du nœud
- `ara.manet.positioning` est composé de classes permettant de simuler le mouvement des nœuds sur un terrain.
- `ara.manet.positioning.strategies` est composé de classes décrivant différentes stratégies de mouvement des nœuds.
- `ara.manet.algorithm.election` composé d'une interface `ElectionProtocol` qui définit les services que doit offrir un nœud implémentant un algorithme d'élection de leader.
- `ara.util` est composée de la classe abstraite `Message` (vu en TP) et d'une classe `Constantes` qui est composé d'un attribut statique pour gérer les affichages via un logger.

Les unités considérées ici sont :

- des millisecondes pour les temps

- des mètres pour les distances
- des mètres par seconde pour les vitesses.

Exercice 1 – Implémentation d'un MANET dans PeerSim

Nous allons construire incrémentalement un modèle de MANET dans PeerSim. En premier lieu vous testerez le code fourni sur le mouvement et le positionnement des nœuds, puis vous coderez un protocole permettant l'émission de messages dans la portée radio d'un nœud, et enfin un protocole permettant de détecter les nœuds voisins directs.

Question 1

En analysant le code de la classe `PositionProtocolImpl`, donnez l'algorithme général de déplacement d'un nœud. Il ne vous est pas demandé de copier/coller le code dans cette question.

Pour coder une politique de déplacement, il faut définir deux types de stratégie :
- une stratégie de placement initial (SPI) sur le terrain,
- une stratégie de déplacement (SD) en choisissant une nouvelle destination.

Les classes du package `manet.positioning.strategies` peuvent décrire seulement une SPI, ou bien décrire seulement une SD, ou bien peuvent décrire les deux si la SPI et la SD sont les mêmes.

Question 2

Testez le simulateur en prenant la stratégie `FullRandom` comme SPI et SD. Le contrôleur graphique sera déclenché toutes les unités de temps, son time_slow pourra être environ de 0.0002. Le seul protocole à renseigner pour ce contrôleur est le PositionProtocol de la simulation, les autres sont pour l'instant optionnels et sans objet. Normalement vous devez voir graphiquement des points verts se déplacer sur l'écran. N'oubliez pas d'amorcer les instances de `PositionProtocol` via un module d'initialisation. Vous répondrez à cette question en donnant le contenu de votre fichier de configuration.

Nous allons ajouter à présent le protocole permettant de simuler l'envoi de messages aux nœuds voisins. À l'émission d'un message (méthode `emit`) vous ajouterez un événement de réception dans la file d'événements du simulateur. Cet événement doit être délivré à tout nœud présent dans la portée de l'émetteur dans `latency` unités de temps sur le protocole de l'emitter, et qui aura la charge de délivrer localement au bon protocole associé au message si et seulement si, il est le destinataire du message ou bien si le message est une diffusion (destinataire = `Emitter.ALL`). On souhaite pouvoir paramétrer le fait que la latence soit strictement fixe (toute transmission de message mettra exactement `latency` unités de temps) soit avec une variance (toute transmission de message mettra en moyenne `latency` unités de temps, impliquant que les canaux ne sont pas FIFO). Pour simuler une variance dans la latence, vous utiliserez la méthode `nextPoisson` de la classe `ExtentedRandom` qui tire une valeur aléatoire selon une loi de Poisson autour de la valeur de latence renseignée. Ainsi dans la méthode `EDSimulator.add` si l'option de la variance est activée dans le fichier de configuration vous renseignerez `CommonState.r.nextPoisson(latency)` au lieu de `latency`.

Question 3

Codez une classe implémentant l'interface `Emitter`. Testez de nouveau avec le moniteur graphique et assurez-vous que les portées sont représentées (cercle en bleu). Vous répondrez à cette question en donnant le code de votre classe.

Nous allons à présent, coder un protocole permettant de détecter les voisins. Ce protocole se basera sur un `Emitter` et implantera la classe `NeighborProtocol` qui impose de maintenir une liste de voisins. Pour détecter la présence des voisins on se basera sur un mécanisme de heartbeat. Les nœuds envoient périodiquement un message `ProbeMessage` dans leur champ d'émission. Tout nœud présent dans le champs d'émission qui reçoit pour la première fois le probe, ajoutera dans la liste, le voisin en question. Dans tous les cas, à chaque réception de probe, le nœud armera un timer pour chaque voisin connu. Si le timer se déclenche, un nœud peut alors considérer que le nœud associé n'est plus son voisin et le supprimera donc de sa liste.

Question 4

Codez une classe implantant l'interface `NeighborProtocol`. Votre classe devra prendre les paramètres de simulation suivants :

- la période d'envoi de message probe
- le temps de déclenchement du timer
- et optionnellement un protocole `NeighborhoodListener`. Ainsi, si ce paramètre est spécifié, la méthode `newNeighborDetected` doit être appelée lors de l'ajout d'un nouveau voisin et la méthode `lostNeighborDetected` doit être appelée lors de la perte d'un voisin.

NB : N'oubliez pas que vous êtes en simulation en temps discrétisé. L'utilisation de Timer Java ou tout autre API utilisant le temps système ne peut pas fonctionner.

Question 5

Testez votre code, et remarquez sur le moniteur graphique l'apparition d'un lien graphique lorsque deux nœuds deviennent voisins.

Question 6

En analysant les codes des classes gérant le positionnement des nœuds qui font appel à un tirage aléatoire, on peut remarquer qu'ils utilisent un objet Random qui leur est dédié (attribut `my_random` initialisé au random de la classe `PositioningConfiguration`). Quelle en est la raison ?

Question 7

Prenez connaissance des différentes stratégies et pour chacune expliquez ce qu'elle fait.

Exercice 2 – Implémentation d'algorithmes d'élection de Leader sur un MANET

Nous allons maintenant implémenter deux algorithmes d'élection de leader dans un MANET. L'algorithme doit assurer qu'à terme, tous les nœuds d'une composante connexe du système reconnaissent le même leader. Le choix d'un leader se fait en général selon un critère définissant un état quantifiable sur chaque nœud (ex : niveau de batterie, puissance de calcul, capacité mémoire, etc.). Dans notre cas, chaque nœud doit être capable de renvoyer une valeur entière qui caractérise cet état (méthode `getValue()` de

l'interface `ElectionProtocol` ici). Dans cette étude, nous considérerons que cette valeur sera calculée aléatoirement sur chaque nœud et restera constante durant toute la durée de l'exécution de l'algorithme. **Les algorithmes doivent donc assurer qu'à terme, tout nœud reconnaisse le nœud ayant la plus haute valeur dans sa composante connexe du réseau.**

Premier algorithme

Le premier algorithme est l'algorithme de Vasudevan-Kurose-Towsley (International Conference on Network Protocols 2004) que l'on nommera *VKT04*. L'article vous est fourni dans les ressources de ce projet. Nous nous intéresserons principalement aux sections III, IV et V.B (il n'est donc pas nécessaire de le lire dans son intégralité) et nous allons procéder à une implémentation progressive de l'algorithme.

Question 1

Dans la section III, expliquez pour chaque hypothèse, pourquoi elle est vérifiée (ou peut être vérifiée) dans notre simulateur.

Question 2

Dans la section IV.A, l'article explique le principe de l'algorithme dans un système statique. Codez dans un premier temps cette implémentation partielle de l'algorithme et testez votre code sur un système statique connexe (choisissez les bonnes stratégies de positionnement identifiées dans l'exercice précédent). Il est attendu une classe `VKT04Statique` qui :

- implémente l'interface `ElectionProtocol`
- implémente l'interface `Monitorable` pour afficher sur le moniteur graphique l'état de chaque nœud : on peut différencier dans cet algorithme trois états (leader inconnu, leader connu, être le leader).
- implémente l'interface `NeighborProtocol`. Puisque nous sommes dans un système statique, nous allons nous passer ici de la couche de détection dynamique de voisins via heartbeat (codée dans le précédent exercice) en calculant directement et statiquement les voisins selon leur position par rapport au rayon d'émission.

Question 3

Une fois que vous vous êtes assuré que la version statique fonctionne, copiez le code précédent dans une classe `VKT04` et implémentez-y la partie dynamique de l'algorithme (section IV.B de l'article).

- Cette classe se basera sur le protocole de détection de voisins. Il faut donc qu'elle implémente en plus l'interface `NeighborhoodListener`. Faites attention de bien considérer les événements d'ajouts et de départ de voisins qui pourraient arriver pendant une élection.
- Dans cette question, nous ne considérons pas encore la détection de perte de lien avec le leader

En testant votre code avec le moniteur graphique, vous devez voir initialement un leader par composante connexe du graphe.

Question 4

Complétez la classe `VKT04` en ajoutant le mécanisme permettant de détecter si un nœud est toujours lié à son leader qui est décrit dans le dernier paragraphe de la section V.B de l'article. En cas de détection de perte de lien avec le leader, une

nouvelle élection sera initiée. Attention : dans la plupart des cas un nœud n'est pas relié directement à son leader, des relais de transmission de messages sont alors nécessaires sur une composante connexe. En testant votre code avec le moniteur graphique, vous devez voir un nouveau leader élu par nouvelle composante connexe du graphe.

Deuxième algorithme

Un second algorithme d'élection de leader basé sur des travaux de recherche en cours au LIP6, vous est proposé en annexe du sujet. Le but de cet algorithme est de permettre à chaque nœud de construire localement et progressivement une vue globale de la topologie du réseau. Le pseudo-code fourni en annexe spécifie le comportement d'un nœud i lors de la réception des événements. Il utilise les heartbeats du MANET précédemment implémentés pour détecter l'état du voisinage de chaque nœud.

Question 5
L'algorithme utilise des horloges logiques. A quoi servent-elles ?
Pourquoi chaque nœud ne peut incrémenter uniquement sa propre horloge ?

Question 6
Pourquoi le knowledge est émis dans sa totalité à la détection de l'arrivée d'un nœud dans le voisinage ?

Question 7
Quel est l'intérêt de créer des edits lors de la déconnexion d'un voisin ou de la réception d'un knowledge, au lieu d'envoyer le knowledge dans son ensemble ?

Question 8
Quel est le contenu d'un edit ?

Question 9
Qu'implique l'adjectif *reachable* ligne 46 ?

Question 10
Implémentez l'algorithme dans PeerSim et vérifiez qu'il fonctionne avec le moniteur graphique.

Question 11
Considérons maintenant qu'il puisse y avoir des pertes de messages suite aux collisions des ondes radio (on ne vous demande pas de les implémenter).

1. Quel impact ceci aurait sur les valeurs des horloges (old_clock et $knowledge[source].clock$) lors des réceptions de edit ?
2. Comment pourrions-nous résoudre efficacement ce problème (encore une fois , il n'est pas demandé de l'implémenter)

Exercice 3 – Étude expérimentale

Dans cet exercice nous allons faire une étude expérimentale de notre système. Nous allons d'abord nous intéresser au comportement du MANET (sans algorithme d'élection),

et ensuite nous ferons une étude comparative des deux algorithmes précédemment implémentés.
Par la suite, toutes nos expériences auront ces paramètres communs :

- nombre de nœuds : 75
- taille de terrain : 1200 x 1200 m
- latence d'émission : 90 msec
- période de probe : 300 millisecondes
- timer de probe : 400 millisecondes

Le temps de simulation sera à définir par vos soins en fonction des besoins de l'expérience.

Étude du MANET

Nous souhaitons étudier l'impact de la portée des nœuds sur la connexité du système. On caractérisera la connexité par le nombre de composantes connexes du système.

Question 1

Codez un contrôleur PeerSim qui logue au moment de son déclenchement (à l'instant t) :

- **la connexité** : le nombre courant de composantes connexes
- **la connexité moyenne** le nombre moyen de composantes connexes depuis le début de l'expérience (c'est à dire la moyenne de toutes les valeurs courantes relevée entre t_0 et l'instant t)
- **la variance de connexité** : l'écart-type associé à la connexité moyenne.

Vous pouvez vous aider des méthodes statiques de l'interface `PositionProtocol` pour calculer les composantes connexes et la classe `peersim.util.IncrementalStats` de l'API de PeerSim pour calculer les moyennes et écart-type.

Question 2

En utilisant les paramètres suivants :

- vitesse min : 5
- vitesse max : 20
- temps de pause : 20 secondes
- portée : à faire varier
- SPI : `FullRandom`
- SD : `FullRandom`

Tracez deux courbes qui prennent en abscisse la portée et en ordonnée :

- la connexité moyenne à t_{end} pour l'une
- la variance de la connexité moyenne à t_{end} pour l'autre

Question 3

Pourquoi est-ce pertinent de choisir la stratégie `FullRandom` comme stratégie de positionnement pour cette expérience ?

Question 4

Interprétez, commentez et expliquez vos résultats.

Étude comparative des algorithmes d'élection de leader

Un algorithme d'élection de leader peut être caractérisé selon deux métriques :

- Le **nombre total de messages** qui ont été envoyés par l'algorithme d'élection (on ne compte pas les messages de hearbeat). Cette métrique permet de connaître le coût en message de l'algorithme.
- Le **taux d'instabilité** qui représente le pourcentage de temps moyen où un nœud possède une valeur de leader erronée ou inconnue. On entend par valeur erronée le fait d'avoir une valeur de leader différente de celle qui est attendue (i.e. le nœud qui a la plus haute valeur dans la composante connexe). Ainsi, pour un pattern de mobilité donné, plus le taux d'instabilité est élevé moins l'algorithme est fiable. Cette métrique à l'instant t se calcule de la manière suivante :

$$TauxInst^t = \frac{\sum_{i=0}^{N-1} Err_i^t}{N.t}$$

  où :
  - $\diamond$ N est le nombre de nœuds du système
  - $\diamond$ Err_i^t est le temps cumulé à l'instant t où le nœud i a eu une valeur de leader erronée ou inconnue.

L'objectif de cette étude est de comparer les deux algorithmes avec ces deux métriques pour plusieurs valeurs de connexité de graphe (que l'on fera varier comme précédemment avec la valeur de la portée).

Question 5

Codez un protocole décorateur d'Emitter qui s'intercalera au-dessus de l'emitter réel et en dessous du protocole d'élection. Ce protocole permettra de manière non intrusive de compter le nombre de messages reçus pour l'algorithme d'élection. On incrémentera un compteur global (statique à cette classe) lors de la réception du message. On ne comptera pas les messages `Probe` du MANET qui font partie de la détection des voisins et de l'élection, de ce fait l'emitter du protocole de détection de voisins sera donc toujours l'emitter réel.

Question 6

Implémentez un mécanisme permettant de calculer le taux d'instabilité.

Question 7

Faites une étude comparative des deux algorithmes selon les deux métriques pour plusieurs valeurs de connexité. Vous ne vous intéresserez qu'aux valeurs des deux métriques à l'instant t_{end} de l'expérience. Des courbes commentées et justifiées sont attendues.

Annexe

Algorithm 1: Global View Leader Election

Local variables of node i **:**
  Typedef **peer** : <id : *int* , value : *int*>
  Typedef **view** : <clock : *int* , neighbors : *set(peer)*>
  knowledge : *array(view)*
Initialisation of node i $(value_i : int)$ **:**
  $knowledge[i].neighbors \leftarrow \{< i,\ value_i >\}$
  $knowledge[i].clock \leftarrow 0$
Upon connected peer j $(value_j : int)$ **:**
  $knowledge[i].neighbors \leftarrow knowledge[i].neighbors \cup \{< j, value_j >\}$
  $knowledge[i].clock \leftarrow knowledge[i].clock + 1$
  Broadcast $(knowledge)$
Upon disconnected peer j **:**
  $edit \leftarrow \{\langle i, \emptyset, \{< j,\ value_j >\}, knowledge[i].clock, knowledge[i].clock + 1\rangle\}$
  $knowledge[i].neighbors \leftarrow knowledge[i].neighbors \setminus \{< j,\ value_j >\}$
  $knowledge[i].clock \leftarrow knowledge[i].clock + 1$
  Broadcast $(edit)$
Upon reception of $knowledge_j$ **from peer** j **:**
  $edit \leftarrow \{\}$
  for ***each*** *peer* p ***in*** $knowledge_j$ **do**
    if $knowledge[p]$ ***is*** *empty* **then**
      $edit \leftarrow edit \cup \{\langle p, p.neighbors, \emptyset, 0, p.clock\rangle\}$
      $knowledge[p].neighbors \leftarrow p.neighbors$
      $knowledge[p].clock \leftarrow p.clock$
    else if $p.clock > knowledge[p].clock$ **then**
      $added \leftarrow p.neighbors \setminus knowledge[p].neighbors$
      $removed \leftarrow knowledge[p].neighbors \setminus p.neighbors$
      $edit \leftarrow edit \cup \{\langle p, added, removed, knowledge[p].clock, p.clock\rangle\}$
      $knowledge[p].neighbors \leftarrow p.neighbors$
      $knowledge[p].clock \leftarrow p.clock$
  if $edit$ ***is not*** *empty* **then**
    Broadcast $(edit)$

```
28 Upon reception of edit from peer j :
29     for each ⟨source, added, removed, old_clock, new_clock⟩ in edit do
30         if added is not empty then
31             if knowledge[source] is empty then
32                 if old_clock equals 0 then
33                     knowledge[source].neighbors ← added
34             else if old_clock equals knowledge[source].clock then
35                 knowledge[source].neighbors ←
                     knowledge[source].neighbors ∪ added
36         if removed is not empty then
37             if knowledge[source] is not empty then
38                 if old_clock equals knowledge[source].clock then
39                     knowledge[source].neighbors ←
                         knowledge[source].neighbors \ removed
40         if knowledge[source] is not empty then
41             if knowledge[source].neighbors was updated then
42                 knowledge[source].clock ← new_clock
43     if knowledge was updated then
44         Broadcast (edit)
45 Upon invocation of Leader() :
46     return id of peer having max value in all reachable peer from i
```