Rapport projet ARA 2017-2018

Michal Rudek, Oskar Viljasaar

13 février 2018

Table des matières

1 Exercice 1 - Implémentation d'un MANET dans PeerSim **2**
1.1 Algorithme de déplacement d'un noeud (Question 1) . . . 2
1.2 Influence des stratégies sur la connexité du graphe (Questions 3, 4, 8) . . . 2
1.3 Impact de la portée sur la densité du graphe (Questions 10 et 11) . . . 3

2 Exercice 2 - Étude de protocoles de diffusion **4**
2.1 Impact du nombre de noeuds sur la densité du graphe (Question 1) . . . 4
2.2 Question 2 (`EmitterCounter`) . . . 5
2.3 Question 4 (`FloodingEmitter`) . . . 5
2.4 Question 5 (`ProbabilisticEmitter`) . . . 6
2.5 Question 6 (`InverseProportionalEmitter`) . . . 7
2.6 Question 7 (`DistanceEmitter`) . . . 8
2.7 Question 8 (`GossipProtocolList`) . . . 8
2.8 Question 9 . . . 10

A Compilation, lancement du code et jeux de test **10**
A.1 `src/Makefile` . . . 10
A.2 Scripts de lancement de simulations . . . 10
A.3 Scripts de traitement de données . . . 10

B Extraits de code **10**
B.1 Fichier de configuration (Exercice 1 question 2) . . . 10
B.2 Implémentation de l'interface `Emitter` (Exercice 1 Question 5) . . . 11
B.3 `DensityController` (Exercice 1 Question 9) . . . 12

— Les codes demandés dans différentes questions du sujet se trouvent dans l'annexe.
— Chaque simulation a été éxecutée avec une différente graine de valeur aléatoire.

1 Exercice 1 - Implémentation d'un MANET dans PeerSim

1.1 Algorithme de déplacement d'un noeud (Question 1)

```
- si le noeud est immobile:
    noeud.vitesse <- rand(vitesse_max)
- si le noeud est en mouvement:
    dest <- next_destination();
    - si (dest.distance > distance_hop):
        aller aussi loin que possible vers le noeud;
    - si noeud.position = destination:
      arrêter le noeud;
    - sinon continuer;
```

L'algorithme utilise le protocole de déplacement suivant : Une valeur de la vitesse est aléatoirement choisie dans l'intervalle [`speed_min`; `speed_max`]. *distance_hop* représente la distance parcourue en une unité de temps. Une fois la destination atteinte, le noeud s'arrête pendant un tic, sinon il boucle en demandeant une nouvelle destination à la stratégie de déplacement.

1.2 Influence des stratégies sur la connexité du graphe (Questions 3, 4, 8)

Strategy1InitNext donne des positions initiales et destinations aléatoires dans le terrain pour chaque noeud.
Strategy3InitNext donne des positions initiales et destinations vers le milieu du terrain, dans un rayon de $\texttt{scope} - \texttt{marge}$, assurant un graphe connexe.
Strategy2Next rend les noeuds immobiles, la connexité du graphe dépend du placement initial des noeuds.
Strategy4Next assume que le graphe est connexe à l'initiation. Elle va déplacer un nœud dans le graphe en s'assurant qu'à la fin, le graphe soit toujours connexe. La connexité du graphe dépend du placement initial des noeuds.
Strategy5Init place les noeuds en haut à droite du terrain, chaque noeud est placé dans le scope d'un autre noeud. Le graphe est initialement connexe.
Strategy6Init place les noeuds en étoile au milieu du terrain, le graphe est donc initialement connexe.

SPI	*SD*	**Connexe**
Strategy1InitNext	Strategy1InitNext	non
Strategy1InitNext	Strategy2Next	non
Strategy1InitNext	**Strategy3InitNext**	**oui**
Strategy1InitNext	Strategy4Next	non
Strategy3InitNext	Strategy1InitNext	non
Strategy3InitNext	Strategy2Next	**oui**
Strategy3InitNext	**Strategy3InitNext**	**oui**
Strategy3InitNext	Strategy4Next	**oui**
Strategy5Init	Strategy1InitNext	non
Strategy5Init	Strategy2Next	**oui**
Strategy5Init	**Strategy3InitNext**	**oui**
Strategy5Init	Strategy4Next	**oui**
Strategy6Init	Strategy1InitNext	non
Strategy6Init	Strategy2Next	**oui**
Strategy6Init	**Strategy3InitNext**	**oui**
Strategy6Init	Strategy4Next	**oui**

TABLE 1 – Impact des différentes SPI et SD sur la connexité du graphe. Les stratégies en gras sont celles assurant la connexité du graphe dans la situation donnée. On s'attend donc à ce qu'une *SPI* en gras utilise une *SD* n'ayant pas d'impact sur la connexité du graphe à l'instant *t=0*.

1.3 Impact de la portée sur la densité du graphe (Questions 10 et 11)

Dans la stratégie 1, l'étendue de la portée a un impact sur la connexité du graphe, la stratégie de déplacement étant celle de choisir des destinations aléatoires dans le terrain. Il est plus facile donc de faire un graphe connexe en prenant une valeur assez grande pour la portée.
La stratégie 3 donnant un graphe connexe dès le début, les noeuds disposent déjà d'un nombre de voisins important. Augmenter la portée pour la stratégie 3 a tendance à légèrement faire diminuer la densité du graphe. Cela peut être expliqué par la distance aléatoire pour la prochaine destination, tirée entre `NextDestinationStrategy.minimum_distance` et $scope - marge$, sachant que $marge$ est plutôt petit (20) et reste constant, alors que la portée peut varier jusqu'à 1000. Le graphe, dans la stratégie 3, est beaucoup plus étendu, et les noeuds peuvent avoir moins d'arcs directs entre eux.

Portee	SPI	SD	D	E/D	ED/D
125	1	1	1.00 +- 0.02	0.27 +- 0.02	0.04 +- 0.00
250	1	1	3.81 +- 0.10	0.14 +- 0.00	0.04 +- 0.00
375	1	1	8.02 +- 0.20	0.13 +- 0.02	0.09 +- 0.02
500	1	1	12.83 +- 0.06	0.11 +- 0.02	0.09 +- 0.02
625	1	1	18.77 +- 0.54	0.11 +- 0.00	0.13 +- 0.01
750	1	1	24.49 +- 0.13	0.10 +- 0.00	0.16 +- 0.02
875	1	1	29.92 +- 0.47	0.09 +- 0.00	0.16 +- 0.02
1000	1	1	35.66 +- 0.44	0.07 +- 0.01	0.11 +- 0.04
125	3	3	29.94 +- 0.25	0.09 +- 0.00	0.17 +- 0.02
250	3	3	26.78 +- 0.27	0.10 +- 0.01	0.21 +- 0.05
375	3	3	25.82 +- 0.41	0.09 +- 0.00	0.16 +- 0.01
500	3	3	25.75 +- 0.58	0.10 +- 0.00	0.15 +- 0.02
625	3	3	25.52 +- 0.33	0.09 +- 0.00	0.15 +- 0.03
750	3	3	25.80 +- 0.23	0.10 +- 0.00	0.16 +- 0.02
875	3	3	25.61 +- 0.47	0.10 +- 0.00	0.16 +- 0.02
1000	3	3	25.21 +- 0.31	0.10 +- 0.00	0.16 +- 0.02

TABLE 2 – Valeurs obtenues pour la question 10, normalisées sur 100 itérations

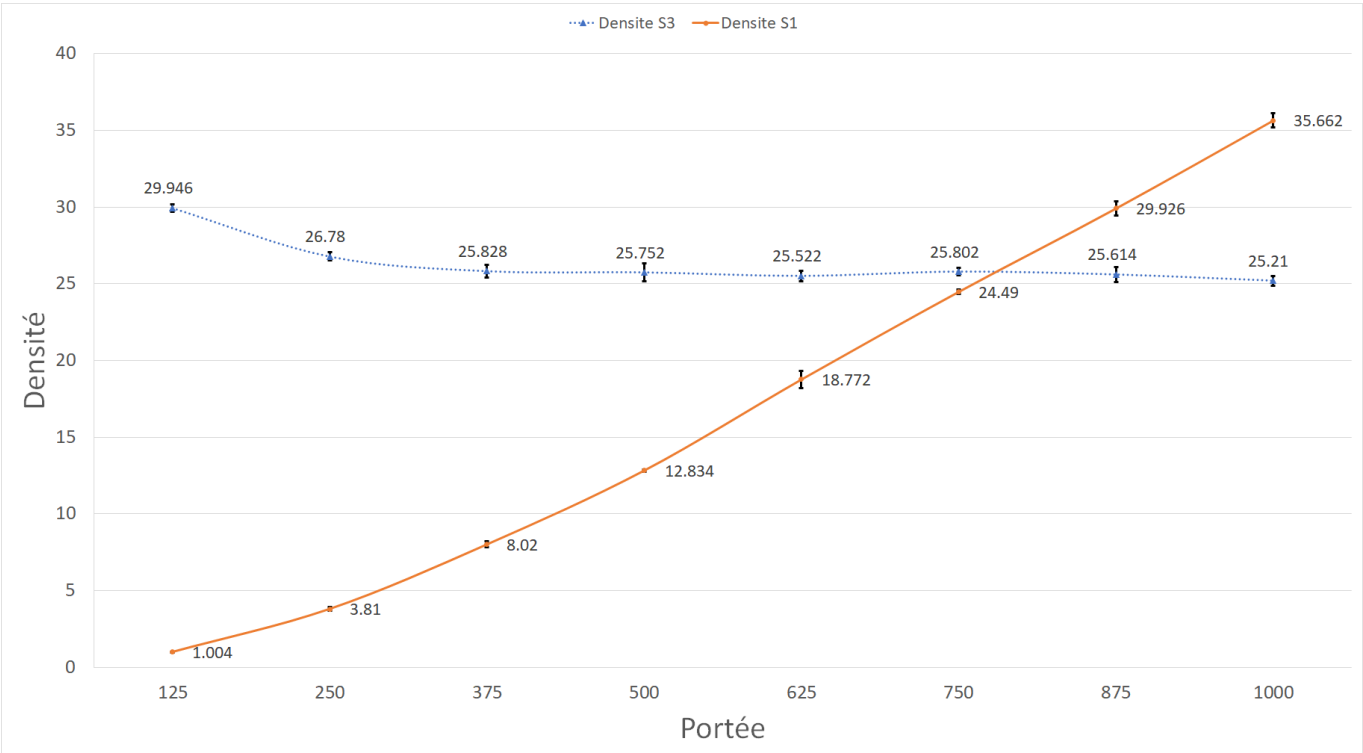

FIGURE 1 – Impact de la portée sur la densité avec la stratégie 1 (orange) et la 3 (bleu).

2 Exercice 2 - Étude de protocoles de diffusion

2.1 Impact du nombre de noeuds sur la densité du graphe (Question 1)

Selon le graphe, le réseau est plutôt chaotique sur un nombre de noeuds faible, allant jusqu'à 50 selon nos résultats. La densité, naturellement, est faible avec un petit nombre de noeuds. L'écart-type entre les différentes valeurs mesurées est fort, les noeuds ont du mal à établir des liens selon les différentes valeurs initiales aléatoires prises. À partir de 50 noeuds, le réseau a un comportement prévisible et la densité croît de manière relativement stable, avec un écart-type faible entre différentes mesures.
En dessous de 50 noeuds, on ne peut donc pas forcément s'attendre à un placement de noeuds idéal, de manière à ce que tous les noeuds soient connectés avec beaucoup de voisins à proximité.

Taille	D-end	ED/D end
10	3.32 +- 0.00	0.26 +- 0.00
20	6.88 +- 1.03	0.55 +- 0.30
30	10.92 +- 2.11	0.28 +- 0.20
40	16.04 +- 4.00	0.17 +- 0.11
50	21.95 +- 4.95	0.08 +- 0.01
60	23.37 +- 6.02	0.13 +- 0.05
70	23.11 +- 4.74	0.08 +- 0.03
80	27.26 +- 1.76	0.07 +- 0.03
90	37.59 +- 3.83	0.06 +- 0.02
100	35.38 +- 3.23	0.04 +- 0.01
120	37.78 +- 5.27	0.03 +- 0.01
140	47.42 +- 12.2	0.04 +- 0.01
160	49.15 +- 10.2	0.03 +- 0.02
180	62.37 +- 5.31	0.03 +- 0.01
200	50.33 +- 7.25	0.02 +- 0.01

TABLE 3 – Valeurs obtenues pour la question 1, normalisés sur 100 itérations

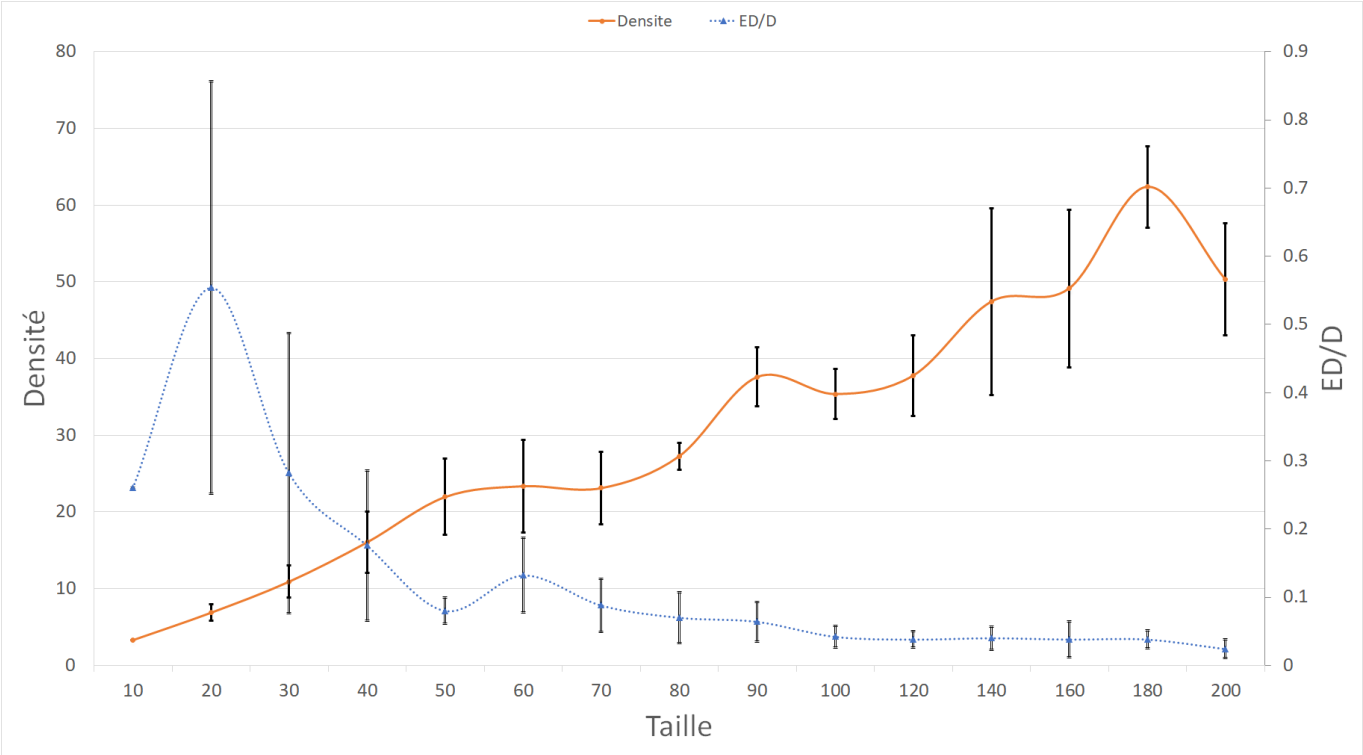

FIGURE 2 – Impact du nombre de noeuds sur la densité et sa variation possible pendant une simulation, avec les stratégies $SPI5$ et $SD4$.

2.2 Question 2 (`EmitterCounter`)

On a défini une classe abstraite `EmitterCounter` utilisant le design pattern *Strategy*, en rendant la méthode `emit` abstraite. Une sous-classe concrète de celle-ci implémente une politique particulière d'émission (`FloodingEmitter`, `ProbabilisticEmitter`, ...) et rend le nombre de messages envoyés selon cette politique. Cette information est destinée à `GossipProtocol` qui se charge de la délivrance (ou non) du message selon si le noeud émetteur se trouve encore dans la portée du récepteur.

On a essayé d'implémenter cette politique de délivrance au niveau de l'émetteur (`EmitterCounter`) en le faisant traiter des réceptions de messages de son protocole encapsulant des messages de n'importe quel protocole au-dessus de lui-même, mais le simulateur rendait trop difficile de détecter une terminaison de manière simple. La fonction `EDSimulator.add(0, ...)` rajoute un évènement à la fin de la file d'éxecution, mais on voulait faire un traitement juste après la délivrance du message par `EmitterCounter`.

2.3 Question 4 (`FloodingEmitter`)

Quelque soit la densité du graphe, tant qu'il est connexe, `FloodingEmitter` assure une atteignabilité de diffusion de 100%. L'économie de rediffusion est naturellement nulle, comme l'émetteur effectue une rediffusion dans tous les cas.

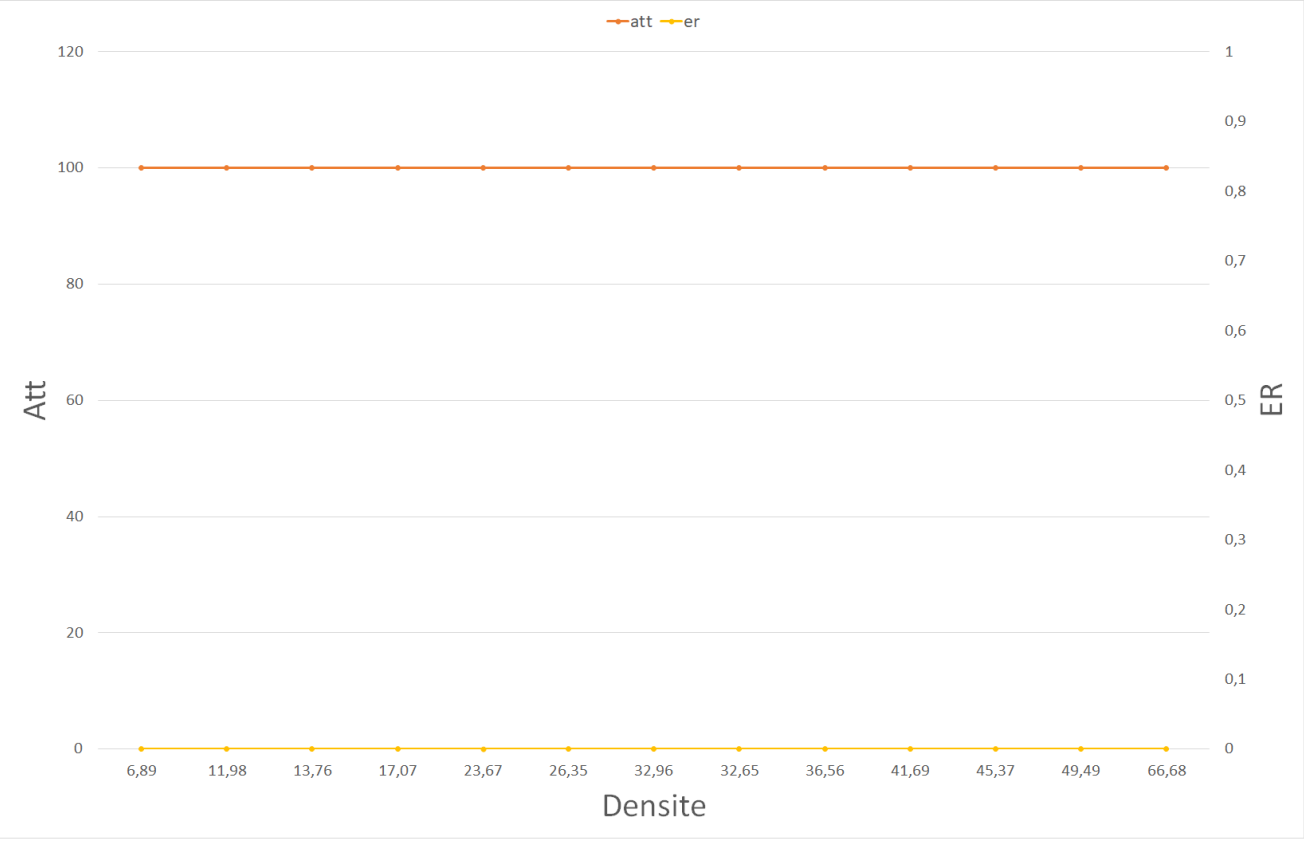

FIGURE 3 – Atteignabilité et économie de rediffusion avec `FloodingEmitter`.

2.4 Question 5 (`ProbabilisticEmitter`)

Le pourcentage de messages reçus augmente clairement selon la probabilité. Selon les résultats expérimentaux, il faut définir une probabilité autour de 0.4 afin d'obtenir une atteignabilité d'au moins 90%, pour une densité moyenne de 4. Par ailleur, à partir de cette densité, l'atteignabilité ne descend jamais en dessous des 80%. Pour obtenir une atteignabilité moyenne du graphe d'au moins 99%, il faudrait utiliser une probabilité d'au moins 0.7 avec une densité d'au moins 6.

Les courbes figurant sur le graphe représentent les différentes classes d'atteignabilité définies dans la question de l'exercice. Les courbes grise et orange montrent qu'il est vite possible d'atteindre la majorité du graphe, même avec une densité faible.

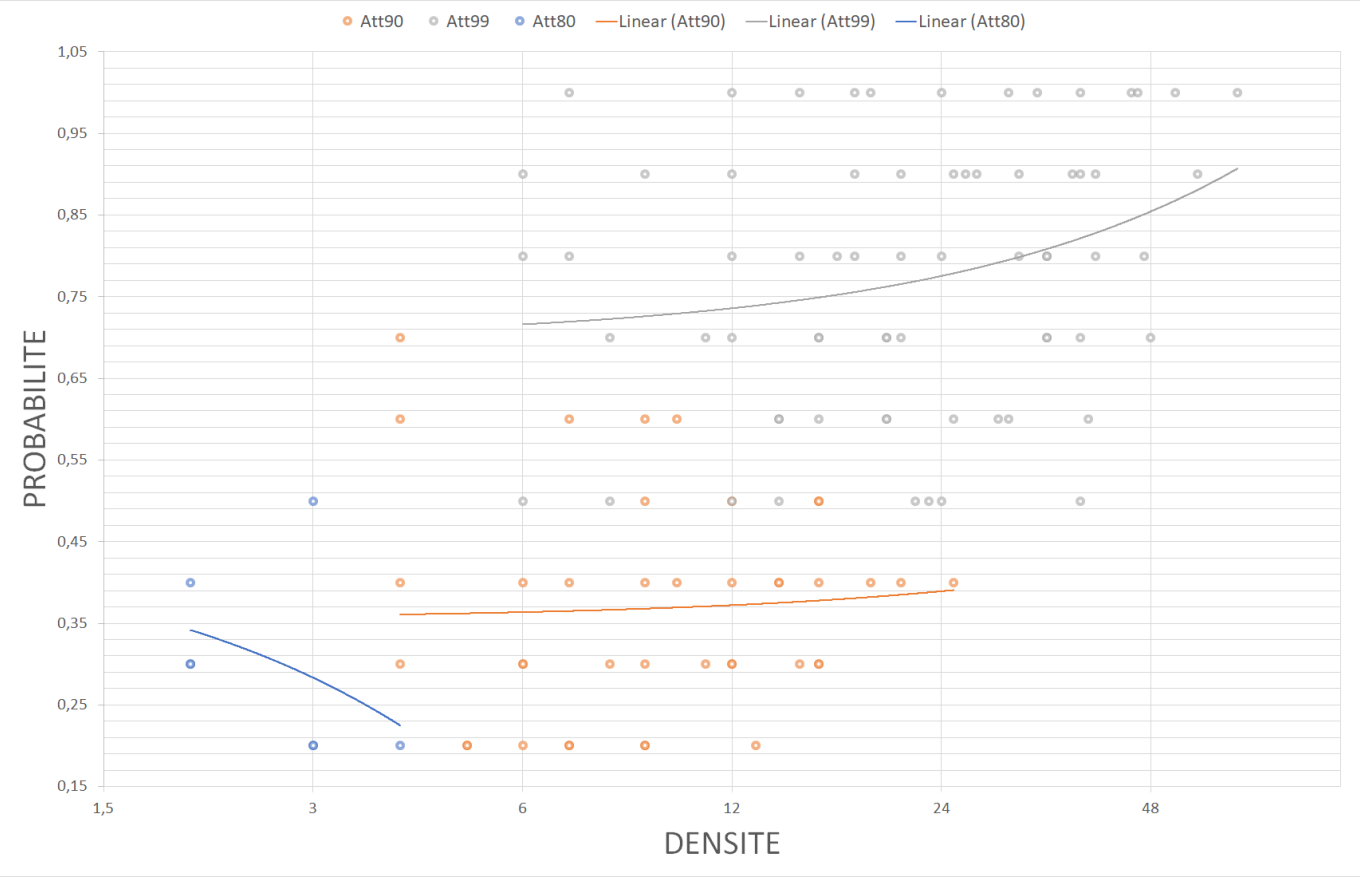

FIGURE 4 – Probabilité en fonction de la densité. Le graphe contient les valeurs de 5 éxecutions différentes pour un même ensemble de valeurs initiales (nombre de noeuds/densité et probabilité).

2.5 Question 6 (`InverseProportionalEmitter`)

On a fait le choix de prendre k comme étant inversement proportionnelle au voisinage. k doit donc varier entre 0 exclus et V inclus. Ici, on a pris $k = 1$, puis on l'incrémente trois fois de $V/4$. Une bonne valeur de k semble se situer entre $V/4$ et $V/2$, pour avoir une atteignabilité forte tout en ayant une certaine économie de rediffusion dans le graphe.

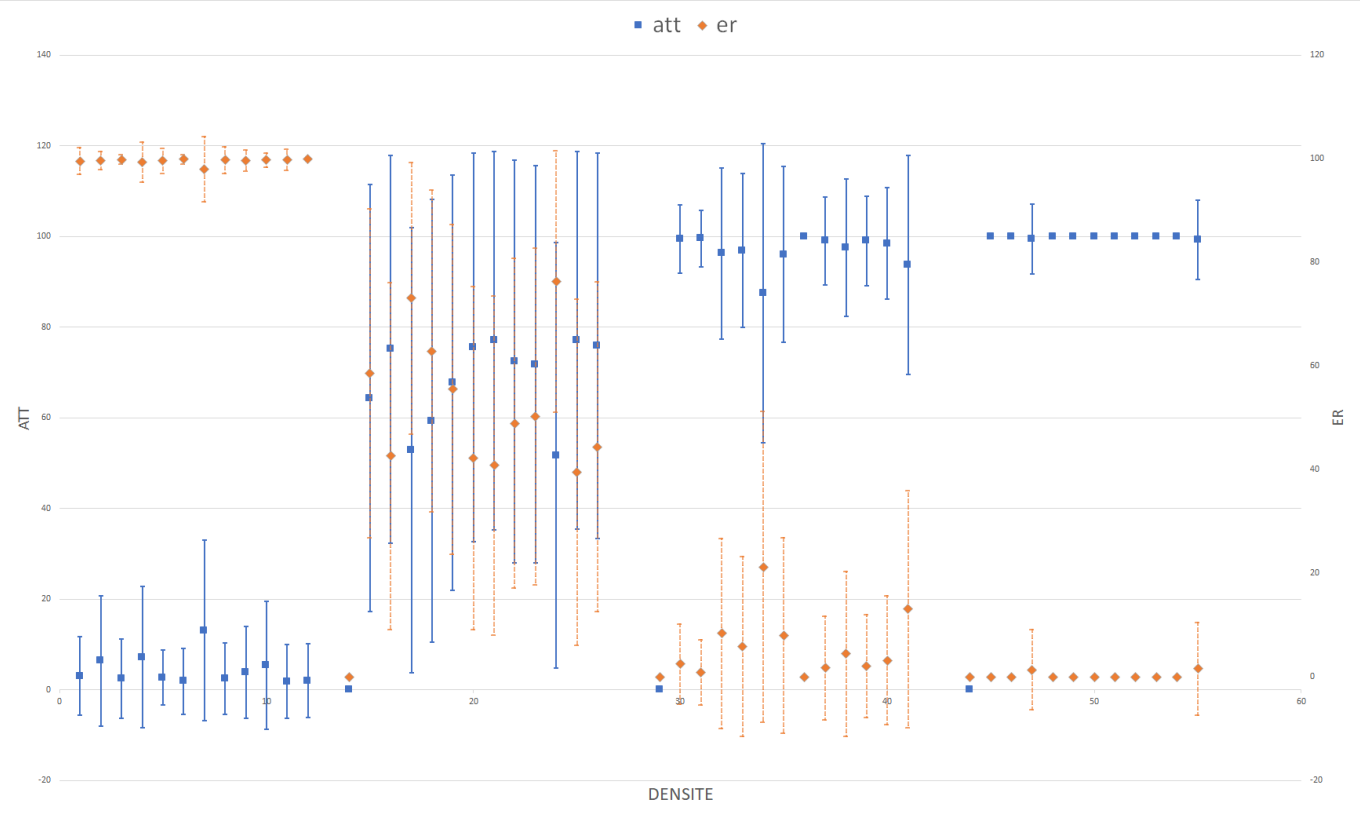

FIGURE 5 – Atteignabilité et économie de rediffusion avec `InverseProportionalEmitter`.

2.6 Question 7 (`DistanceEmitter`)

Dans le code, l'émetteur `DistanceEmitter` rediffuse un message selon la probabilité p donnée dans l'énoncé : plus l'émetteur est loin du récepteur, plus le récepteur a donc de chance de rediffuser le message dans sa portée.
Les résultats d'une diffusion utilisant `DistanceEmitter` nous montrent qu'il est possible d'atteindre la majorité des noeuds avec une économie de rediffusion oscillant entre 30 et 40%. Les expériences étant éxecutées à chaque fois avec une différente graine aléatoire, l'atteignabilité d'un message peut varier selon le placement parfois inopportun de certains noeuds.

Noeuds	*Att*	*EAtt*	*ER*	*EER*	*D*
20	97.34	9.23	37.09	13.27	7.37
30	99.38	5.02	32.60	10.38	12.22
40	97.87	7.43	33.33	9.43	14.02
50	98.75	5.71	31.14	8.26	15.94
60	99.99	0.15	31.74	6.79	25.59
70	99.92	0.25	30.46	6.19	27.11
80	99.53	3.53	32.23	7.67	33.61
90	99.59	4.17	29.79	6.39	29.05
100	99.89	0.94	29.60	5.35	36.56
120	100.00	0.01	28.98	4.91	41.84
140	99.95	0.89	29.34	4.98	44.81
160	99.98	0.10	28.27	4.33	50.14
200	99.98	0.08	28.78	4.14	67.65

TABLE 4 – Valeurs obtenues pour la question 7, normalisés sur 100 itérations

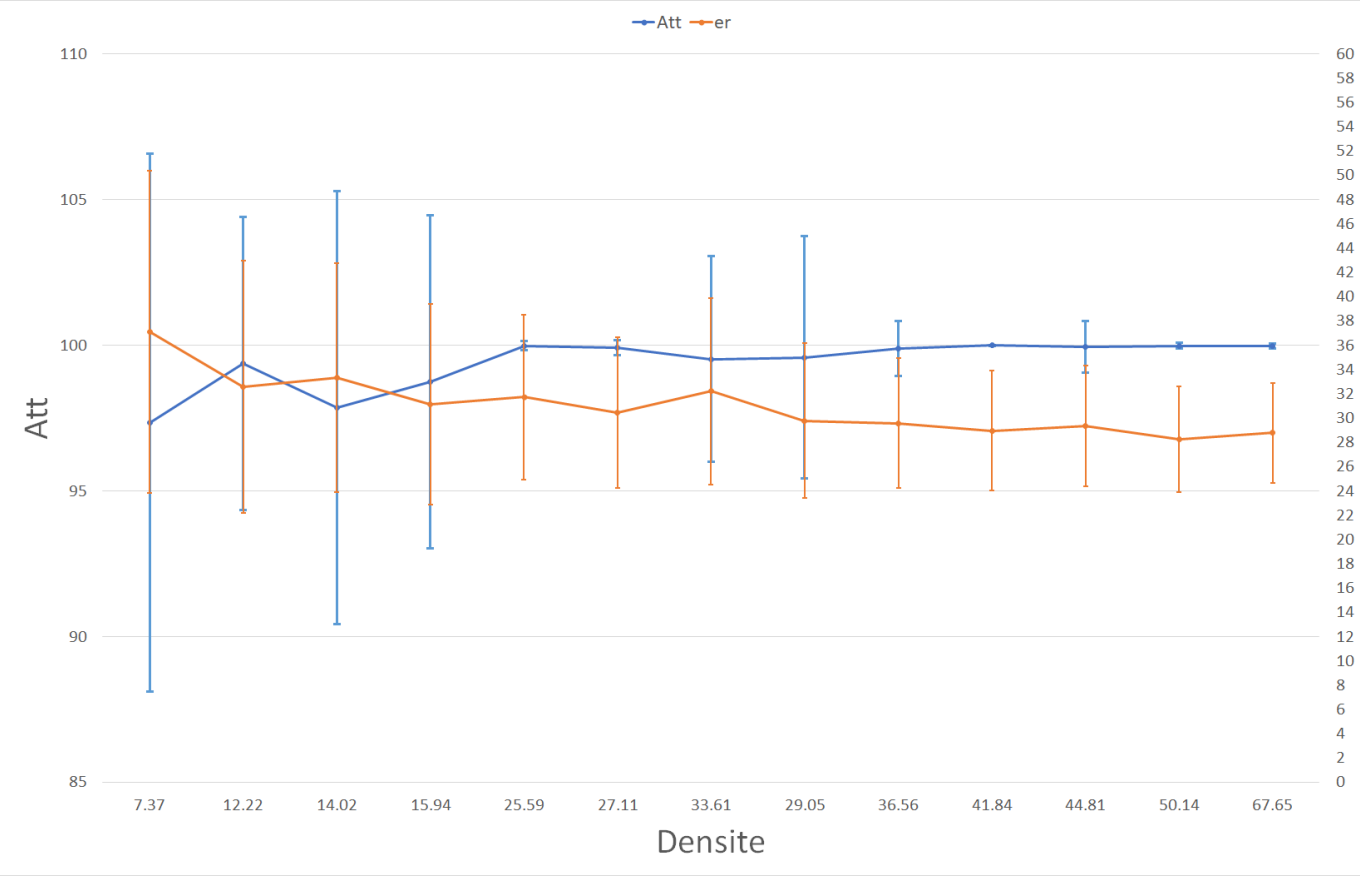

FIGURE 6 – Atteignabilité et économie de rediffusion de `GossipProtocolImpl` utilisant `DistanceEmitter`.

2.7 Question 8 (`GossipProtocolList`)

La classe `GossipProtocolList` a été implémentée de manière à ce qu'à la première *non*-transmission d'un message, le message est ré-émis **dans tous les cas** après le déclenchement d'un timer à durée aléatoire entre `timer_min` et `timer_max`. De ce fait, nous atteignons des valeurs d'atteignabilité importantes (100% le plus souvent avec `DistanceEmitter`). Cela semble logique, en prenant compte qu'on prend `Strategy5Init` et `Strategy4Next` comme stratégies de placement initial et de mouvement, ce qui nous assure un graphe connexe durant toute la durée de l'expérience.

L'économie de rediffusion baisse légèrement en fonction de la densité selon nos résultats expérimentaux. En effet, plus la densité augments, plus un noeud a une chance d'avoir déjà reçu le message, il est donc moins utile de rediffuser le message. Cela étant dit, l'*ER* ne baisse pas de manière forte selon la densité ; l'algorithme de détection de voisinage

n'ayant pas reçu de message se prouve plutôt efficace dans ce cas de figure.

InverseProportionalEmitter avec GossipProtocolList nous assure 100% d'atteignabilité à tout moment, ce qui nous assure une forte économie de rediffusion sur des valeurs de k faibles.

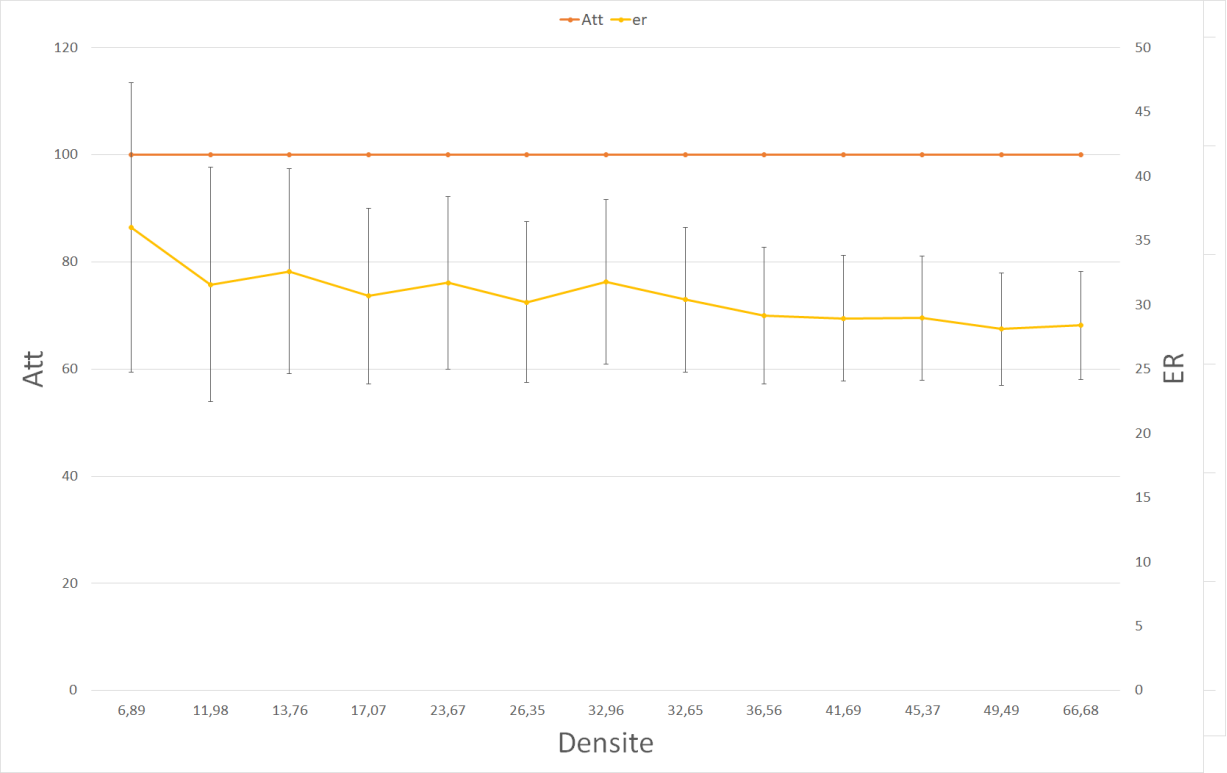

FIGURE 7 – Atteignabilité et économie de rediffusion de `DistanceEmitter` combiné avec la détection du voisinage n'ayant pas reçu les messages (`GossipProtocolList`).

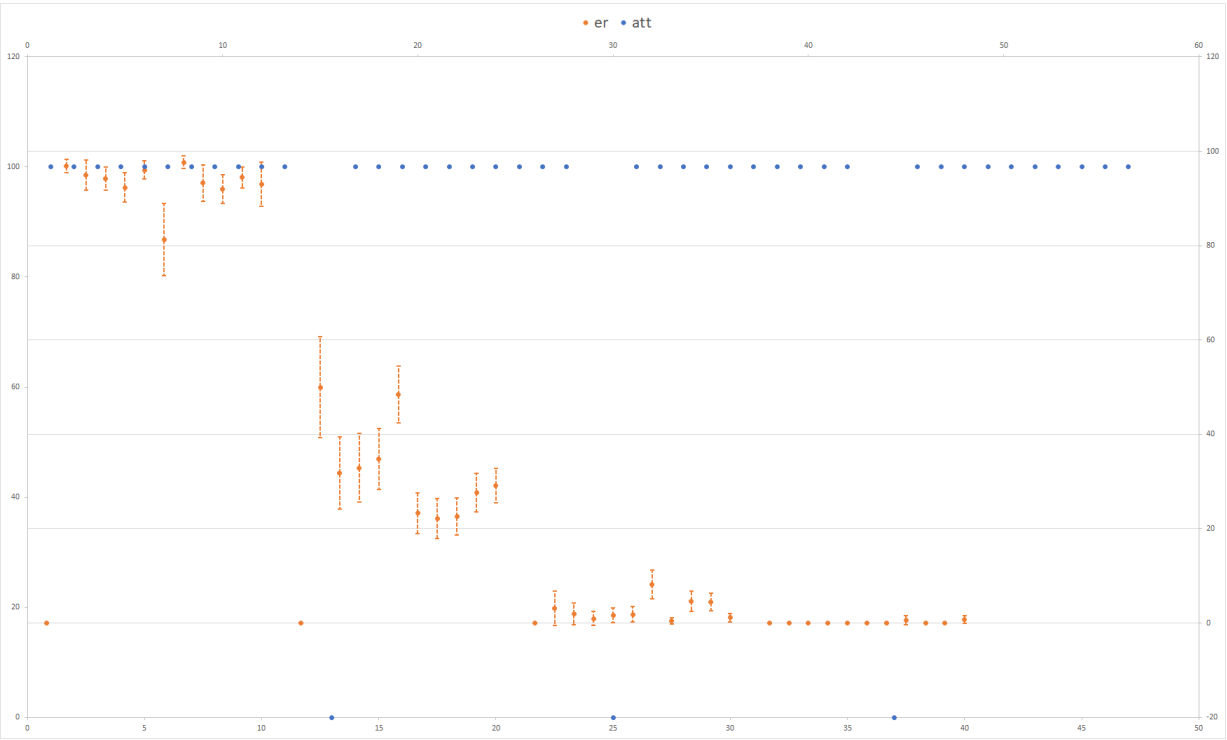

FIGURE 8 – Atteignabilité et économie de rediffusion de `InverseProportionalEmitter` combiné avec la détection du voisinage n'ayant pas reçu les messages (`GossipProtocolList`).

2.8 Question 9

L'algorithme de détection de messages potentiellement non-reçus dans le voisinage améliore grandement l'atteignabilité des algorithmes de diffusion testés ci-dessus. L'émetteur `DistanceEmitter` a tendance à marcher sur toutes les densités, on pourrait utiliser `InverseProportionalEmitter` sur des faibles densités à condition d'utiliser la détection de messages non-reçus.

A Compilation, lancement du code et jeux de test

A.1 `src/Makefile`

Un fichier `Makefile` est inclus dans le dossier `src`. Celui-ci reconnaît les directives :
- `make` compile le projet.
- `make run` lance une instance de simulation telle que spécifiée dans `src/manet/cfg_initial.txt`.
- `make clean` nettoie le projet des compilés.
- `make bench_clean` nettoie le dossier `src` des dossiers de résultats des benchmarks.

Le `Makefile` admet par ailleurs deux variables :
- `DIR_PEERSIM=<chemin>` : le dossier d'installation de Peersim, qu'il faudra **obligatoirement** soit modifier, soit spécifier dans `make` et `make run`.
- `CFG=<chemin>` : le chemin d'un fichier de configuration Peersim, initialisé à `src/manet/cfg_initial.txt` par défaut.

A.2 Scripts de lancement de simulations

Une série de scripts `perl` sont fournis avec le projet, prenant en argument le chemin d'installation Peersim, fournissant une manière rapide de simuler plusieurs expériences.

Les scripts créent un dossier `resultats/bench_<exercice>_<date>/` où seront stockés les résultats utiles pour la question. Le dossier contiendra un fichier de configuration pour chaque chaque expériences, et un autre fichier `.results` contenant les résultats des simulations sur plusieurs itérations. Chaque expérience est éxecutée avec une graine aléatoire différente.

Les scripts de l'exercice 2 (hormis la question 1) sortent un fichier par exécutions, de forme :

```
$ cat results/bench_ex2q5_2018-02-02-23:00:20/cfg_bench_30_0.2_0.results
70,34;26,44;0,18;0,29
2,45;0,19;0,06
```

Les valeurs de la première ligne représentent l'atteignabilité, son écart-type et l'économie de rediffusion respectivement. (`GossipController`)

Les valeurs de la deuxième ligne représentent la densité, son écart-type et le pourcentage respectif à ces valeurs. (`DensityController`)

A.3 Scripts de traitement de données

exemple : `bench_ex2.py <chemin_results>`

Ces scripts calculent les moyennes et écarts-type sur les fichiers résultats fournis par les scripts de lancement de simulation. Les donées produites par ces scripts sont utilisés pour les représentations graphiques des résultats de simulation.

B Extraits de code

B.1 Fichier de configuration (Exercice 1 question 2)

```
simulation.endtime 50000
random.seed 5
network.size 10
init.initialisation Initialisation
control.graph GraphicalMonitor
control.graph.positionprotocol position
control.graph.time_slow 0.0002
control.graph.step 1
```

B.2 Implémentation de l'interface `Emitter` (Exercice 1 Question 5)

```java
package manet.communication;

import manet.Message;
import manet.positioning.Position;
import manet.positioning.PositionProtocol;
import peersim.config.Configuration;
import peersim.core.Network;
import peersim.core.Node;
import peersim.core.Protocol;
import peersim.edsim.EDSimulator;

public class EmitterImpl implements Emitter {

    private int latency;
    private int scope;
    protected int this_pid;
    private int position_protocol;

    public static int messa;


    /**
     * Standard peersim constructor
     * @param prefix
     */
    public EmitterImpl(String prefix) {
        String tmp[]=prefix.split("\\.");
        this_pid=Configuration.lookupPid(tmp[tmp.length-1]);
        //System.out.println(prefix + "." + PAR_POSITIONPROTOCOL);
        this.position_protocol=Configuration.getPid(prefix+"."+PAR_POSITIONPROTOCOL);
        this.latency = Configuration.getInt(prefix + "." + PAR_LATENCY);
        this.scope = Configuration.getInt(prefix + "." + PAR_SCOPE);
    }

    /**
     * Copy constructor
     * @param latency latency
     * @param scope scope
     * @param position_protocol position protocol
     */
    public EmitterImpl(int latency, int scope, int position_protocol) {
        this.latency = latency;
        this.scope = scope;
        this.position_protocol = position_protocol;
    }


    /**
     * This method sends the messages
     * @param host The current host
     * @param msg The message it's sending
     */
    @Override
    public void emit(Node host, Message msg) {
        PositionProtocol prot = (PositionProtocol) host.getProtocol(position_protocol);
        for (int i=0; i < Network.size(); i++) {
            Node n = Network.get(i);
            PositionProtocol prot2 = (PositionProtocol) n.getProtocol(position_protocol)
            double dist = prot.getCurrentPosition().distance(prot2.getCurrentPosition())
            if (dist < scope && n.getID() != host.getID()) {
                if (msg.getIdDest() == -1)
```

```
                    EDSimulator.add(latency,
                            new Message(msg.getIdSrc(), n.getID(), msg.getTag(), msg.get
                            n, msg.getPid());
                }
            }

    }

    @Override
    public int getLatency() {
        return latency;
    }

    @Override
    public int getScope() {
        return scope;
    }

    @Override
    public Object clone(){
        EmitterImpl res=null;
        try {
            res=(EmitterImpl)super.clone();
        } catch (CloneNotSupportedException e) {}
        return res;
    }
}
```

B.3 DensityController (Exercice 1 Question 9)

```
package manet;

import manet.detection.NeighborProtocol;
import peersim.config.Configuration;
import peersim.core.CommonState;
import peersim.core.Control;
import peersim.core.Network;

import java.util.ArrayList;

public class DensityController implements Control {


    private static final String PAR_NEIGHBOR = "neighbours";
    private static final String PAR_VERBOSE = "verbose";
    private static final String PAR_STEP = "step";

    private final int this_pid;

    private int verbose = 0; // Est-ce que l'on print les resultats sur stdout
    private int step; // step du controlleur

    private double
            dit = 0.0,  // la moyenne du nombre de voisins par noeud a l'instant t (dens
            eit = 0.0,  // l'ecart-type de dit (donc a l'instant t)
            dt  = 0.0,  // densite moyenne sur le temps (avg of d\_dt)
            et  = 0.0,  // disparite moyenne de densite sur le temps (avg of d\_et)
            edt = 0.0;  // variation de la densite au cours du temps (ecart type des val
    // donc de toute la sim jusqu'a mtn)

    // Arrays containing data for dt, et and edt calculations
    private ArrayList<Double>
            d_dt  = new ArrayList<Double>(), // Updated by dit()
```

```
34              d_et  = new ArrayList<Double>(), // Updated by eit()
35              d_edt = new ArrayList<Double>(); // Updates by edt()
36
37      public DensityController(String prefix) {
38          this.this_pid = Configuration.getPid(prefix+"."+PAR_NEIGHBOR);
39          this.verbose = Configuration.getInt(prefix + "." + PAR_VERBOSE);
40          this.step = Configuration.getInt(prefix + "." + PAR_STEP);
41      }
42
43
44      @Override
45      // @return true if the simulation has to be stopped, false otherwise.
46      public boolean execute() {
47          // Over-time averages
48          dt = dt();
49          et = et();
50          edt = edt();
51
52          // 'Live' values
53          dit = dit();
54          eit = eit();
55
56  //        if (this.verbose != 0)
57              if (CommonState.getTime() >= CommonState.getEndTime()-step)
58                  printCols();
59
60          return false;
61      }
62
63      /** A l'instant T **/
64
65      /**
66       * Calculates the average number of neighbours in the
67       * network when called (works on 'live' data)
68       * D_i(t) : Moyenne du nombre de voisins par noeud a l'instant t
69       *
70       * Updates dit and d_dt[]
71       *
72       * @return double average neighbors per node
73       */
74      private double dit() {
75          double
76                  sum = 0.0,
77                  avg = 0.0;
78
79          for (int i = 0 ; i < Network.size() ; i++) {
80              double n_neigs = ((NeighborProtocol) Network.get(i).getProtocol(this_pid)).g
81
82              sum += n_neigs;
83          }
84
85          avg = sum / Network.size();
86          d_dt.add(avg);  // Add to history
87          return avg;
88      }
89
90      /**
91       * Calculates the standard deviation
92       * E_i(t) : L'ecart type de D_i(t) (dit())
93       * Works on 'live' data
94       * Updates eit and d_et[]
95       *
96       * @return l'ecart-type de dit
```

```
97          */
98      public double eit() {
99          double stdDev = 0.0;
100         for (Double d : d_dt) {
101             if (this.verbose != 0)
102                 System.out.format("d: %.2f\n", d);
103             stdDev += Math.pow(d - dit, 2);
104         }
105         if (this.verbose != 0)
106             System.out.format("Stddev: %.2f ", stdDev);
107         double avg = stdDev/d_dt.size();
108         stdDev = Math.sqrt(avg);
109         if (this.verbose != 0)
110             System.out.format("avg: %.2f stddev: %.2f\n", avg, stdDev);
111         d_et.add(stdDev);   // Add to history
112         return stdDev;
113     }
114
115
116
117     /** Stats for all until current **/
118
119     /**
120      * La moyenne de l'ensemble des valeurs D_i(t') pour tout t' < t
121      * donc densite moyenne sur le temps
122      *
123      * Updates dt, works with history array
124      *
125      * @return average density so far
126      */
127     public double dt() {
128         double sum = 0.0, ret = 0.;
129         if (!d_dt.isEmpty()) {
130             for (Double d : d_dt)
131                 sum += d;
132             ret = sum / d_dt.size();
133         }
134         return ret;
135     }
136
137     /**
138      * La moyenne de l'ensemble des valeurs E_i(t') pour tout t' < t
139      * donc disparite moyenne de densite sur le temps
140      *
141      * Updates et, works with history array
142      *
143      * @return average density so far
144      */
145     public double et() {
146         double sum = 0.0, ret = 0.;
147         if (!d_et.isEmpty()) {
148             for (Double d : d_et)
149                 sum += d;
150             ret = sum / d_et.size();
151         }
152         return ret;
153     }
154
155     /**
156      * L'ecart type des valeurs D_i(t'), pour tout t' <= t, ce qui
157      * permet de juger de la variation de la densite au cours du temps.
158      * Plus le @return de cette fonction est elevee par rapport au resultat
159      * de et(), plus le reseau a change de densite moyenne au cours
```

```
160         * du temps.
161         *
162         * Updates recalculates etd, works with history array
163         *
164         * @return
165         */
166        public double edt() {
167            double stdDev = 0.0;
168            if (!d_dt.isEmpty()) {
169                for (Double d : d_dt)
170                    stdDev += Math.pow(dt - d, 2);
171                stdDev = stdDev / d_dt.size();
172            }
173            d_edt.add(stdDev);
174            return stdDev;
175        }
176
177
178        /** We're lazy so functions for q10
179         * Col1 = D(t=end)
180         * Col2 = E(t=end) / D(t=end)
181         * Col3 = ED(t=end) / D(t=end)
182         * **/
183        public double col1() { return getDt(); }
184        public double col2() { return (getEt() / getDt()); }
185        public double col3() { return (getEdt() / getDt()); }
186
187        public void printCols() {
188            String s = String.format("%.2f;%.2f;%.2f", col1(), col2(), col3());
189            System.out.println(s);
190            System.out.flush();
191
192            if (verbose != 0)
193                System.err.println("Should have flushed");
194        }
195
196
197
198        /* Getters */
199        public double getEdt() { return edt; }
200        public double getEt() { return et; }
201        public double getDt() { return dt; }
202        public double getEit() { return eit; }
203        public double getDit() { return dit; }
204
205    }
```