Enoncé du projet du cours de Réseaux

— Simulation d'un protocole de routage à états de liens —

B. Quoitin et M. Michel

23 novembre 2015

1 Objectifs

L'objectif de ce projet est de renforcer votre compréhension du fonctionnement des protocoles de routage présentés dans le chapitre 4 du cours. Ce projet se focalise en particulier sur les protocoles de type à états de liens (*link state*).

Afin d'atteindre l'objectif du projet, il vous est demandé de réaliser une implémentation d'un protocole de routage à états de liens. Votre implémentation sera réalisée dans un simulateur conçu spécialement pour le TP et qui sera décrit succintement dans ce document. Votre implémentation devra supporter les trois phases d'un protocole de routage à états de liens, à savoir, la découverte des voisins, l'inondation (*flooding*) et le calcul des plus courts chemins à l'aide de l'algorithme de Dijkstra.

La Section 2 décrit les contraintes techniques qui vous sont imposées pour la réalisation de votre implémentation. La Section 3 décrit succintement le fonctionnement du simulateur et de son interface de programmation. Cette dernière est illustrée par un exemple complet d'implémentation d'un protocole de routage de type à vecteurs de distance. Finalement, la Section 4 décrit les documents à remettre à l'issue du projet.

2 Contraintes techniques

2.1 Identification d'un routeur

Chaque routeur doit posséder un identifiant unique : le *router-ID*. Cet identifiant permet d'identifier le routeur d'origine de plusieurs messages même si ces messages sont envoyés avec des adresses IP source différentes.

L'identifiant d'un routeur sera dérivé automatiquement des adresses IP possédées par le routeur : l'adresse de valeur numérique la plus élevée sera choisie comme identifiant.

2.2 Echange de messages entre routeurs voisins

Les messages échangés entre les routeurs ne peuvent pas utiliser le forwarding IP car cela nécessiterait que les routes IP soient déjà établies. Or c'est précisément le rôle du protocole de routage d'établir ces routes.

Le protocole qui sera implémenté enverra des messages aux routeurs adjacents en les encapsulant dans des datagrammes IP envoyés en broadcast, i.e. vers l'adresse `255.255.255.255`. Les datagrammes seront envoyés directement à travers les interfaces, sans effectuer de recherche dans la table de forwarding.

2.3 Découverte des voisins

La phase de découverte des voisins permet à un routeur de découvrir l'existence de ses voisins ainsi que leur identifiant. Cette phase se repose typiquement sur l'envoi périodique de messages

Hello. Dans un message *Hello*, un routeur indique son identifiant ainsi que la liste des identifiants des voisins déjà découverts. Un routeur A considère qu'il a une adjacence avec un voisin B s'il a reçu un message *Hello* de B et que sa propre adresse figure dans la liste de voisins du message de B. La découverte des voisins est illustrée à la Figure 1.

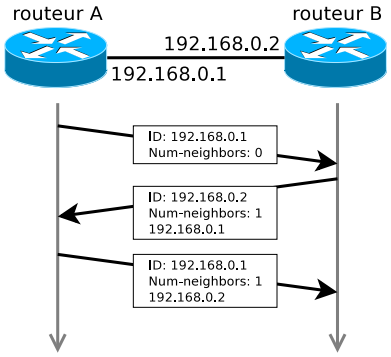


FIGURE 1 – Découverte des voisins.

Le format d'un message *Hello* est illustré à la Figure 2. Les champs du message sont les suivants :

— Identifiant du routeur : adresse IP identifiant le routeur à l'origine de ce LSP (voir Section 2.1). Il s'agit d'une adresse IP encodée en binaire dans un champ de 32 bits.

— Liste des voisins : liste des identifiants des voisins. La liste est encodée comme suit. Premièrement, un champ de 8 bits spécifie le nombre de voisins listés. Deuxièmement, pour chaque voisin, un champ de 32 bits indique son *router-ID* (cf. Section 2.1).

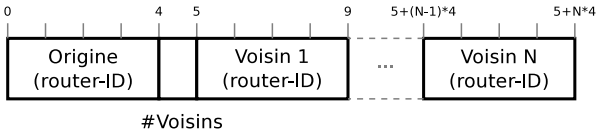


FIGURE 2 – Format d'un message *Hello*.

La fréquence d'envoi des messages *Hello* doit pouvoir être spécifiée à chaque routeur, lors de l'instanciation du protocole.

2.4 Format des LSP et de la LSDB

Dans la phase d'inondation (*flooding*) les routeurs devront être capables de s'échanger des *Link State Packets* (LSP) qui permettront à chacun de construire une représentation du réseau complet.

Le format d'un LSP est illustré à la Figure 3. Les champs d'un LSP sont les suivants :

— Identifiant du routeur : adresse IP identifiant le routeur à l'origine de ce LSP (voir Section 2.1). Il s'agit d'une adresse IP encodée en binaire dans un champ de 32 bits.

— Numéro de séquence : numéro de séquence de ce LSP. Il s'agit d'un nombre entier positif encodé en binaire dans un champ de 32 bits.

— Liste des adjacences : liste des adjacences du routeur. La liste des adjacences est encodée comme suit. Premièrement, le nombre d'adjacences est indiqué sous la forme d'un entier encodé sur 8 bits. Deuxièmement, pour chaque adjacence, l'identifiant du routeur voisin ainsi que le coût du lien vers ce voisin sont spécifiés. L'identifiant du voisin est son *router-ID* encodé sous la forme d'un entier de 32 bits. Le coût vers le voisin est également encodé sous la forme d'un entier de 32 bits.

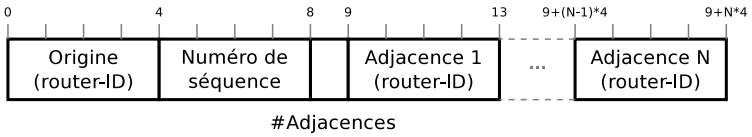


FIGURE 3 – Format d'un *Link State Packet* (LSP).

Chaque routeur devra stocker les LSP reçus dans une *Link State DataBase* (LSDB), une petite base de données conservée en mémoire. Vous êtes libres de choisir quelle structure de données employer pour implémenter cette LSDB. Il ne vous est pas demandé de supprimer les LSP reçus au bout d'un certain temps (*ageing*). Vous pouvez néanmoins supporter optionnellement cette fonctionnalité.

La fréquence d'envoi des messages *LSP* en secondes doit pouvoir être spécifiée à chaque routeur, lors de l'instanciation du protocole.

2.5 Coûts des liens

Le coût des liens est une valeur entière positive dans l'intervalle $[0, 2147483647]$. La valeur maximale 2147483647 est considérée comme l'infini. Cette valeur est définie comme la constante `Integer.MAX_VALUE` en java.

Le coût des liens doit pouvoir être spécifié lors de la construction de la topologie du réseau. Cette fonctionnalité est supportée par le simulateur lorsqu'il charge une topologie de réseau complète à partir d'un fichier (cf. Section 3.1.1).

Votre implémentation doit pouvoir supporter le changement de coût d'un lien en cours de simulation.

2.6 Implémentation de l'algorithme de Dijkstra

Bien que le cours théorique ait brièvement discuté de la complexité algorithmique de l'algorithme de Dijkstra en fonction des structures de données utilisées pour implémenter la file de priorité, il ne vous est pas demandé d'utiliser la structure de données la plus efficace. Vous êtes libres du choix de cette structure de données.

S'il existe plusieurs chemins de coût minimum, votre implémentation doit en choisir un seul de façon déterministe.

3 Fonctionnement du simulateur

Le simulateur de réseaux à utiliser pour réaliser ce projet se compose de deux parties : un ordonnanceur (*scheduler*) et un modèle de réseaux.

L'ordonnanceur se charge de gérer les évènements de la simulation. Des exemples typiques de tels évènements sont l'envoi d'un message et l'expiration d'un *timer*. Pour illustrer l'usage du simulateur, considérons l'envoi par un noeud au temps t d'un message sur un lien de communication. Ce message est délégué par le noeud au simulateur avec le temps de propagation δt sur le lien. Le simulateur se charge de délivrer le message envoyé au noeud qui se trouve de l'autre côté du lien au temps $t + \delta t$. Pour réaliser cela, l'ordonnanceur appelle une méthode de réception du message (*callback* ou *listener*) implémentée par le noeud destinataire.

Le modèle de réseaux fourni avec le simulateur permet de représenter un ensemble d'équipements réseaux que nous appelerons des noeuds ainsi que des liens de communications entre les noeuds. Les noeuds peuvent être des hôtes et des routeurs. Le modèle de réseaux permet également la représentation d'interfaces de communication physiques et virtuelles, des trames échangées entre interfaces, ainsi que le support d'une couche de communication similaire à IP comportant la définition de datagrammes, le support d'une table de forwarding et le traitement de datagrammes.

La Figure 4 donne un aperçu du diagramme de classes du simulateur. L'ordonnanceur prend la forme de la classe `Scheduler`. Un réseau est un ensemble de noeuds et est modélisé par la classe `Network`. Un noeud, modélisé par la classe `Node`, contient de multiples interfaces physiques. Une interface physique est modélisée par l'interface `HardwareInterface`. Une interface physique a un nom tel que "`eth0`" pour une interface Ethernet. Le modèle ne comprend actuellement qu'une unique implémentation d'une interface physique : l'interface Ethernet modélisée par la classe `EthernetInterface`. Une interface Ethernet possède une adresse Ethernet représentée par une instance d'`EthernetAddress`. Un hôte est un noeud sur lequel il est possible de faire fonctionner plusieurs applications. Un hôte est modélisé par la classe `Host` et une application par la classe `Application`.

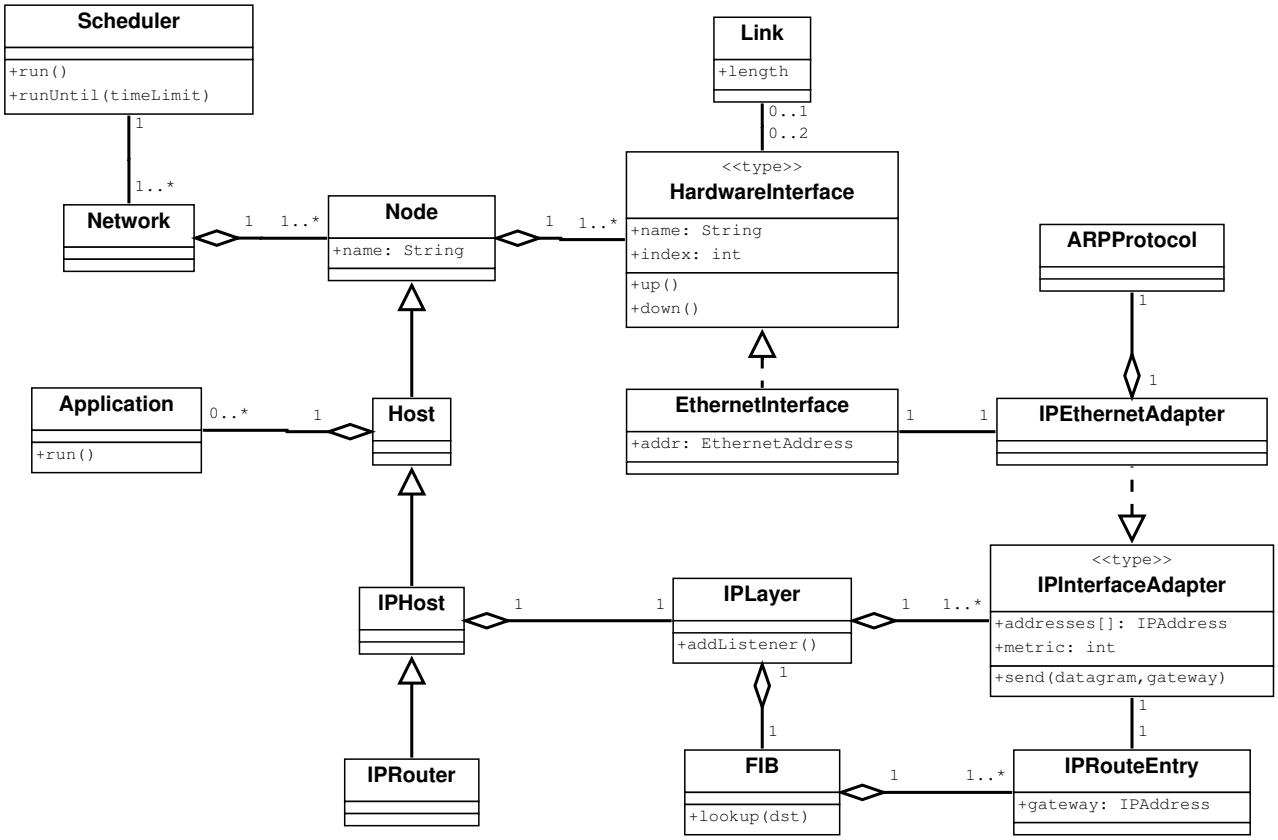


FIGURE 4 – Diagramme de classes du modèle de réseaux.

Le support du protocole réseau IP est assuré par la classe `IPLayer`. Un hôte qui supporte IP est modélisé par la classe `IPHost`. Un routeur IP est modélisé par la classe `IPRouter`. La couche IP maintient une liste d'interfaces par lesquelles il est possible d'envoyer/recevoir des datagrammes IP. L'interface `IPInterfaceAdapter` représente une interface de communication IP. Une telle interface peut être liée à une interface physique, comme c'est le cas pour la classe `IPEthernetAdapter` qui fait le lien entre IP et une interface Ethernet. Cette classe fait en outre appel au protocole ARP modélisé par la classe `ARPProtocol` pour assurer la correspondance entre les adresses IP et les adresses Ethernet. La couche IP supporte également des interfaces loopback modélisées par la classe `IPLoopbackAdapter` (non montré sur la figure). Finalement, la couche réseau contient également une table de forwarding modélisée par la classe `FIB`. La FIB contient de multiples entrées modélisées par la classe `IPRouteEntry`. Chaque entrée de la FIB contient une

adresse destination[1] , une interface (IP) de sortie et éventuellement l'adresse IP d'une passerelle (*gateway*).

Afin d'être complet, la Figure 5 présente le diagramme des classes utilisées pour modéliser différents types de messages. L'interface `Message` est à la base de la hiérarchie des classes message. L'interface `MessageWithPayload` modélise un message qui en transporte un autre. Cette interface définit un champ `type` qui indique la nature du *payload* transporté. La classe `EthernetFrame` représente une trame Ethernet. Cette dernière peut transporter des messages de différents types (notamment des datagrammes IP et des messages ARP). La classe `Datagram` représente un datagramme IP. Cette dernière peut transporter des messages de différents types (typiquement de la couche transport[2]). La classe `ARPMessage` représente les messages utilisés par le protocole ARP.

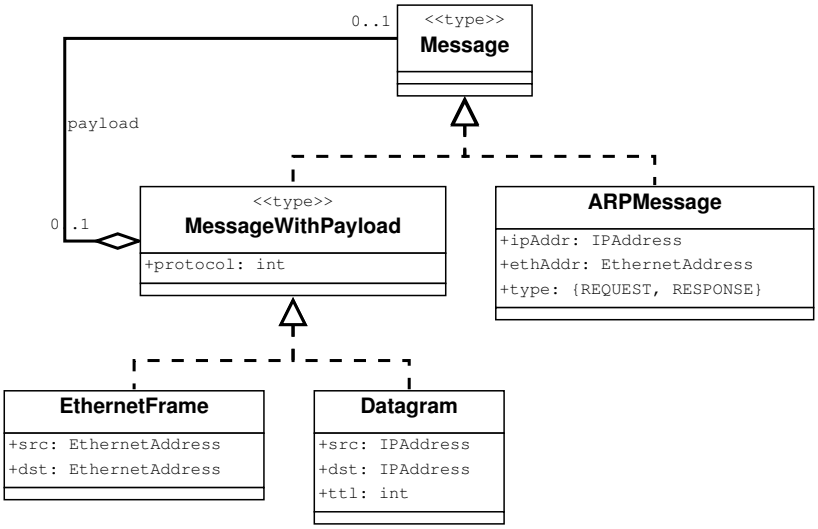


FIGURE 5 – Diagramme de classes des messages.

3.1 Interface de programmation

3.1.1 Chargement d'une topologie existante

Afin d'exécuter le protocole de routage, celui-ci devra être déployé sur une topologie composée de plusieurs routeurs et liens. Le simulateur permet de charger à partir d'un seul fichier texte une topologie complète composée de multiples routeurs et liens. Le simulateur se charge d'instancier à votre place les routeurs et liens correspondant. Le code suivant illustre comment il est possible d'instancier un réseau complet en seulement 2-3 lignes.

```
0 String filename= ...
1 AbstractScheduler scheduler= new Scheduler();
2 Network network= NetworkBuilder.loadTopology(filename, scheduler);
```

La syntaxe utilisée pour décrire textuellement la topologie du réseau est relativement simple. La Figure 6 donne un exemple d'une topologie simple composée de 3 routeurs et 4 liens ainsi que de la représentation textuelle correspondante. Chaque ligne du fichier permet d'effectuer une déclaration. Cinq types de déclarations sont possibles :

1. Contrairement à la réalité, le modèle d'IP implémenté dans ce simulateur n'effectue pas une recherche de type *longest-matching* mais bien un *exact-matching* sur base de l'adresse destination. La notion de sous-réseau IP est également absente du modèle dans sa version actuelle.
2. Le modèle actuel ne représente pas les protocoles de la couche transport.

— `router` déclare un routeur. Le paramètre spécifie le nom du routeur.
— `link` déclare un lien entre deux interfaces. Les paramètres spécifient le nom du premier routeur et de son interface, puis le nom du second routeur et de son interface, et finalement la longueur du lien en mètres. A ce stade, il n'est possible de relier ensemble que des interfaces Ethernet (`eth`).
— `lo` déclare une interface *loopback* dans le routeur déclaré précédemment. Le paramètre spécifie l'adresse IP attribuée à l'interface.
— `eth` déclare une interface Ethernet dans le routeur déclaré précédemment. Le paramètre spécifie l'adresse IP attribuée à l'interface.
— `metric` déclare le coût du lien déclaré précédemment. La déclaration accepte 1 ou 2 paramètres. S'il n'y a qu'un paramètre, celui-ci est le coût assigné au lien dans les deux directions. S'il y a deux paramètres, le premier (resp. le second) est le coût assigné au lien dans la direction *forward* (resp. *backward*) du lien. Il est possible de remplacer l'un ou les deux paramètres de la déclaration par le caractère '`?`'. Dans ce cas, la métrique correspondante sera remplacée par la valeur maximale (`Integer.MAX_VALUE`) et sera considérée comme infinie.

```
router R1
    lo 10.0.0.1
    eth 192.168.0.1
    eth 192.168.1.1
    eth 192.168.3.1
router R2
    lo 10.0.0.2
    eth 192.168.0.2
    eth 192.168.1.2
    eth 192.168.2.1
router R3
    lo 10.0.0.3
    eth 192.168.2.2
    eth 192.168.3.2
link R1 eth0 R2 eth0 5000000
    metric 1 2
link R1 eth1 R2 eth1 5000000
    metric 2 1
link R1 eth2 R3 eth1 5000000
    metric 10
link R2 eth2 R3 eth0 5000000
    metric ? 5
```

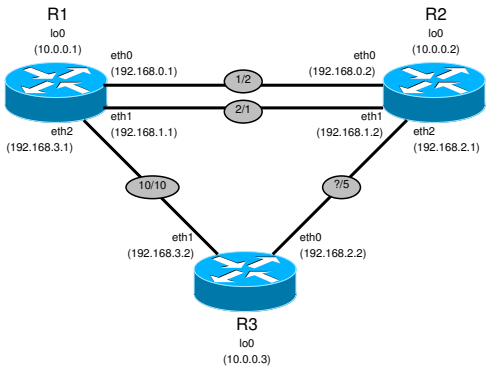


FIGURE 6 – Exemple de topologie et sa représentation textuelle.

3.1.2 Ajout d'applications/protocoles à un noeud

Le protocole à implémenter sera modélisé comme une application. Cette application devra être ajoutée à chacun des routeurs (`IPRouter`) du réseau. Afin d'ajouter une application à un hôte, il suffit d'utiliser la méthode `addApplication`. Cette méthode prend un seul paramètre qui est une instance de l'application à ajouter.

Le code suivant illustre comment ajouter une application `DVRoutingProtocol` à chacun des routeurs de la simulation.

```
0 for (Node n: network.getNodes()) {
1   if (!(n instanceof IPRouter))
2     continue;
3   IPRouter router= (IPRouter) n;
4   router.addApplication(new DVRoutingProtocol(router, true));
5   router.start();
6 }
```

L'appel de la méthode `router.start()` a pour effet de démarrer toutes les applications actuellement associées au routeur, en appelant leur méthode `start()`. Le second argument du construc-

teur de la classe `DVRoutingProtocol` utilisé dans l'exemple ci-dessus indique si le routeur annonce ou non ses propres destinations locales à travers le protocole de routage.

3.1.3 Envoi de datagrammes

Afin d'envoyer un datagramme via une interface particulière, il suffit d'utiliser la méthode `send` de la classe `IPInterface` en lui passant deux paramètres : une instance de la classe `Datagram` et éventuellement l'adresse IP du routeur *gateway* auquel le datagramme doit être transmis. Dans le cas de l'envoi en *broadcast*, i.e. vers l'adresse `255.255.255.255`, le *gateway* ne doit pas être spécifié (`null`).

Pour créer un datagramme, il suffit d'utiliser le constructeur de la classe `Datagram`. Celui-ci prend 5 paramètres : les adresses IP source et destination de type `IPAddress`, un entier identifiant le protocole, le TTL initial (de type `byte`) et le *payload* de type `Message`. Dans l'exemple ci-dessous, le datagramme est envoyé à l'adresse broadcast et le *payload* est un message *Hello*.

```
0 IPInterface iface= ...
1 Datagram datagram= new Datagram(iface.getAddress(), IPAddress.BROADCAST,
2                                 IP_PROTO_LS, 1, hello);
3 iface.send(datagram, null);
```

3.1.4 Réception de datagrammes

Afin de recevoir les datagrammes qui lui sont destinés, le protocole de routage utilisera la primitive `addListener` de la classe `IPHost`. En paramètre de `addListener`, il est nécessaire de fournir le numéro du protocole de routage et une implémentation de l'interface `IPInterfaceListener`.

Dans l'exemple ci-dessous, le programme s'enregistre pour recevoir tous les datagrammes dont le numéro de protocole est égal à `IP_PROTO_LS` et qui sont destinés à la machine locale.

```
0 IPInterfaceListener listener= new IPInterfaceListener() {
1   public void receive(IPInterface src, Datagram datagram) {
2     System.out.println("Datagram␣received:␣"+datagram);
3   }
4 }
5 IPHost ip= ...
6 ip.addListener(IP_PROTO_LS, listener);
```

3.1.5 Utilisation d'un timer

La classe `AbstractTimer` permet d'exécuter une action après un intervalle de temps donné ou de répéter une action à intervalle donné. Comme son nom l'indique, la classe `AbstractTimer` est abstraite, ce qui siginifie qu'il est nécessaire de d'abord en dériver une classe concrète qui implémente la méthode `run`. L'exemple suivant illustre la création d'une classe `MyTimer` descendant d'`AbstractTimer` et qui affiche à intervalle régulier le temps actuel de la simulation.

```
0 private class MyTimer extends AbstractTimer {
1   public MyTimer(AbstractScheduler scheduler, int interval) {
2     super(scheduler, interval, true);
3   }
4   public void run() throws Exception {
5     System.out.println("Current␣time:␣"+scheduler.getCurrentTime());
6   }
7 }
8 AbstractTimer timer= new MyTimer(scheduler, 1);
9 timer.start();
```

Notez que si vous créez un timer qui se répète indéfiniment (en passant `true` comme valeur de l'argument `repeat` du constructeur du timer), le timer va constamment ajouter des évènements à l'ordonnanceur et celui-ci ne se terminera jamais.

3.1.6 Lancement de la simulation

Afin de lancer la simulation et de traiter les évènements en attente, la méthode `run` de l'ordonnanceur (instance de `Scheduler`) doit être appelée. La méthode retournera lorsque la file d'évènements du simulateur sera vidée.

Attention ! il est possible que la méthode `run` ne se termine jamais si des évènements exécutés par le simulateur ajoutent eux-mêmes de nouveaux évènements dans la file de l'ordonnanceur. Pour cette raison, l'ordonnanceur possède également une méthode `runUntil` à laquelle un temps limite d'exécution est passé en argument. Le temps limite est un temps simulé.

3.1.7 Manipulation de la FIB

Les routes calculées par le protocole de routage sur un routeur peuvent être installées dans la FIB de celui-ci. Elles sont alors utilisables pour le *forwarding* IP. La classe `IPLayer` permet l'ajout et la suppression d'entrées dans la FIB par l'intermédiaire des méthodes `addRoute` et `removeRoute`. La méthode `addRoute` prend un unique argument : une instance de la classe `IPRouteEntry`. La méthode `removeRoute` supprime la route dont la destination est fournie en argument.

L'exemple suivant illustre comment une route peut être ajoutée à la FIB. Le troisième paramètre du constructeur d'`IPRouteEntry` est une chaîne de caractères qui identifie l'origine de la route. Pour les routes ajoutées statiquement, l'identifiant est "`static`". Pour les routes provenant d'un protocole de routage, il peut s'agir du nom du protocole (p.ex. "`dv-routing`").

```
0 IPAddress dst= IPAddress.getByAddress(192, 168, 0, 1);
1 IPInterfaceAdapter oif= ...
2 IPRouteEntry re= new IPRouteEntry(dst, oif, IP_PROTO_NAME);
3 ip.addRoute(re);
```

La couche IP permet également de lister l'ensemble des routes contenues dans la FIB. La méthode `getRoutes` est prévue à cet effet. L'exemple suivant illustre comment récupérer et afficher pour chaque routeur l'ensemble de ses routes.

```
0 for (Node n: network.getNodes()) {
1   if (!(n instanceof IPRouter))
2     continue;
3   IPRouter router= (IPRouter) n;
4   System.out.println("Router [" + router.name + "]");
5   for (IPRouteEntry re: router.getIPLayer().getRoutes())
6     System.out.println("\t" + re);
7 }
```

3.1.8 Surveillance de l'état des interfaces

Un protocole de routage doit surveiller l'état des interfaces réseaux afin de pouvoir mettre à jour les routes calculées en conséquence. D'une part, il est nécessaire de surveiller si une interface est active, i.e. si elle peut envoyer/recevoir des messages. D'autre part, il est nécessaire de surveiller le coût associé à une interface car celui-ci peut être modifié par l'opérateur du réseau.

Une application peut s'enregistrer auprès de n'importe laquelle des interfaces de son noeud hôte. Si l'état ou le coût de l'interface changent, l'application en sera ainsi avertie. Le mécanisme utilisé est un mécanisme *listener*. Le code suivant illustre ce mécanisme : un *listener* implémentant l'interface `InterfaceAttrListener` est enregistré auprès d'une interface. Lorsqu'un attribut de l'interface est modifié, la méthode `attrChanged` est appelée.

```
InterfaceAttrListener listener= new InterfaceAttrListener() {
  public void attrChanged(Interface iface, String attr) {
    System.out.println("iface=" + iface + " : attr=" + attr +
      " value=" + iface.getAttribute(attr));
  }
};
Interface iface= ...
iface.addAttrListener(listener);
```

Toute interface supporte l'attribut `STATE` de type `Boolean` qui indique si l'interface est active ou non. L'attribut `STATE` peut être modifié par les méthodes `up` et `down` qui permettent respectivement d'activer et de désactiver l'interface.

Une interface IP (`IPInterfaceAdapter` et descendantes) supporte également l'attribute `METRIC` de type `Integer` qui représente le coût du lien dans la direction partant de l'interface. La méthode `setMetric` permet de modifier le coût de l'interface.

La méthode `getAttribute` permet de récupérer la valeur actuelle de n'importe quel attribut supporté par une interface.

3.1.9 Etat du modèle

Afin de documenter et de déboguer les applications et protocoles développés, le simulateur permet de générer des représentations graphiques (1) de la topologie du réseau et (2) des routes vers une destination donnée actuellement installées dans la FIB des routeurs.

La classe `NetworkGrapher` permet de générer ces représentations graphiques en collaboration avec l'outil graphviz [3] développé par AT&T. `NetworkGrapher` génère une description textuelle du graphe dans le langage "dot". Cette description textuelle est enregistrée dans un fichier. Le fichier peut alors être passé à graphviz pour générer une image du graphe dans des formats classiques tels que `png`, `pdf` et `eps`.

L'exemple ci-dessous illustre comment `NetworkGrapher` peut être utilisé pour générer une représentation graphique de la topologie d'un réseau.

```
File f= new File("/tmp/topology.graphviz");
Writer w= new BufferedWriter(new FileWriter(f));
NetworkGrapher.toGraphviz(network, new PrintWriter(w));
w.close();
```

Une fois le fichier texte généré par `NetworkGrapher`, il faut le passer à graphviz pour générer l'image correspondante. La commande suivante permet de générer le graphique montré à la Figure 7(a) en utilisant la commande `neato` (partie de graphviz).

```
bash$ cat /tmp/topology.graphviz | neato -Tpdf > topology.pdf
```

La classe `NetworkGrapher` permet également d'afficher sur le graphe du réseau les routes installées dans la FIB des routeurs pour une destination donnée. Pour cela, la méthode `toGraphviz2` est utilisée. Les Figures 7(b) à 7(f) montrent les routes calculées par le protocole de routage à vecteurs de distance décrit à la Section 3.2. Le protocole de routage a calculé les routes vers l'adresse de l'interface *loopback* de chaque routeur. Chaque figure concerne une destination différente : la destination est identifiée avec un double cercle. Un arc muni d'une flèche de A vers B indique que A a une entrée dans sa FIB qui passe par B pour joindre la destination considérée. Par exemple, sur la Figure 7(e) relative à la destination R4 (`10.0.0.4`), l'arc allant du routeur R2 à R1 en sortant par l'interface `eth3` indique que le routeur R2 a une route vers `10.0.0.4` dont l'interface de sortie est `eth3`.

3. L'outil graphviz est disponible gratuitement à l'adresse `http://www.graphviz.org`.

3.2 Exemple : protocole de routage de type DV

Le package `reso.examples` contient plusieurs exemples d'utilisation du simulateur. Vous êtes invités à consulter le code source de ces exemples pour vous familiariser avec le simulateur. Cette section décrit l'exemple du package `reso.examples.dv_routing` qui vise à implémenter une version simplifiée d'un protocole de routage à vecteurs de distance. L'exemple est découpé en 4 classes différentes : `Demo`, `DVMessage`, `DVRoutingEntry` et `DVRoutingProtocol`.

Le point d'entrée du programme est situé dans la méthode `main` de la classe `Demo`. Cette méthode se charge de charger une topologie, comme décrit en Section 3.1.1. La méthode se charge ensuite d'ajouter le protocole de routage à chaque routeur de la topologie, comme décrit en Section 3.1.2. La méthode lance alors la simulation en invocant la méthode `run` de l'ordonnanceur, comme décrit en Section 3.1.6. Lorsque la simulation est terminée, la méthode affiche pour chaque routeur l'ensemble de ses routes.

Le protocole est implémenté dans la classe `DVRoutingProtocol`. Le protocole démarre par l'exécution de la méthode `start`. Cette méthode se charge premièrement de s'enregistrer comme *listener* pour les datagrammes portant le numéro de protocole `IP_PROTO_DV` (cf. Section 3.1.4). Le numéro de protocole `IP_PROTO_DV` est alloué automatiquement par la méthode `allocateProtocolNumber` de la classe `Datagram`. La seconde action de la méthode `start` est l'envoi du premier vecteur de distance à travers chacune de ses interfaces (cf. Section 3.1.3). Les vecteurs de distance sont envoyés en *broadcast* par l'intermédiaire de chaque interface IP (sauf via les interfaces *loopback*). Lors de la réception d'un vecteur de distance d'un autre routeur, chaque destination annoncée est traitée. Le coût annoncé pour cette distance est comparé au coût éventuellement déjà connu. Si le coût est amélioré, le routeur met à jour sa FIB (cf. Section 3.1.7) et envoie un vecteur de distances mis à jour à tous ses voisins.

La classe `DVMessage` représente un message vecteur de distances. Un tel message comprend une liste de couples (destination, distance).

La classe `DVRoutingEntry` étend la classe `IPRoutingEntry` et est utilisée pour associer des informations supplémentaires aux routes stockées dans la FIB (cf. Section 3.1.7).

4 Délivrables

Ce projet est à réaliser par **groupe de deux étudiants**. Le projet doit être terminé pour le **23 decembre 2015** . **Au plus tard** à cette date, à 12h, vous devez avoir rendu les délivrables du projet : un rapport ainsi que le code source et une version compilée de votre implémentation. Les délivrables sont à poster sur moodle. Une fois la deadline passée plus aucun projet ne sera accepté. Veuillez suivre scrupuleusement les consignes indiquées ci-dessous.

Le rapport doit être **exclusivement** fourni au **format PDF**. Le rapport ne doit pas faire plus de **5 pages**. Le rapport doit décrire brièvement l'approche utilisée dans votre implémentation, les difficultés éventuellement rencontrées et l'état de l'implémentation finale. Le début du rapport doit contenir une section intitulée "Construction et exécution" qui décrit comment compiler et exécuter votre implémentation. Cette section ne doit contenir qu'un paragraphe d'au plus 5 lignes. Votre rapport doit mentionner clairement en dessous du titre les noms, prénoms et matricules de chaque membre du groupe ainsi que le numéro du groupe.

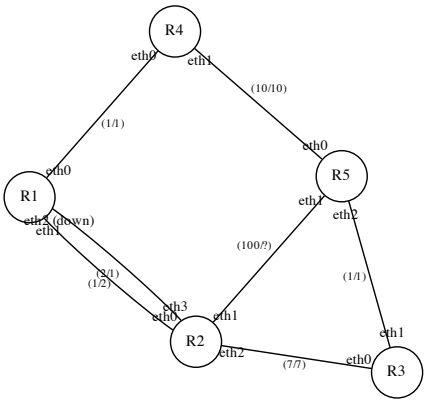
Les sources et binaires à fournir sont uniquement ceux que vous avez développés vous-mêmes. Inutile de nous fournir les sources/binaires du simulateur, nous les avons déjà. **Vous ne devez pas modifier le code du simulateur**. Si vous pensez devoir modifier le simulateur veuillez d'abord en discuter avec nous.

Le rapport, les sources (.java) et la version compilée (.class) de votre implémentation devront être fournis **dans une archive**. L'archive devra être nommée "`tp-reseaux-clr-2015-gr`X" où X est remplacé par le numéro de votre groupe. Votre archive sera fournie au format "zip" ou "tar.gz". Votre archive doit respecter l'arborescence de répertoires suivante :

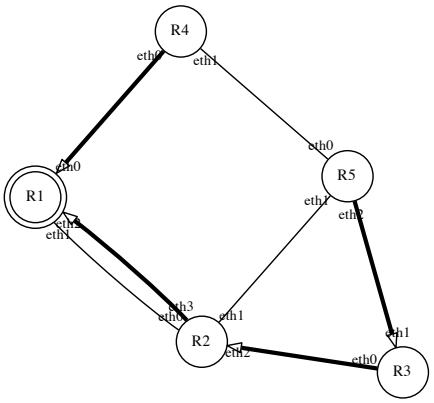
```
tp-reseaux-clr-2015-grX/
  rapport.pdf
```

```
src/
  ensemble des fichiers source (.java)
build/
  ensemble des fichiers compilés (.class)
```

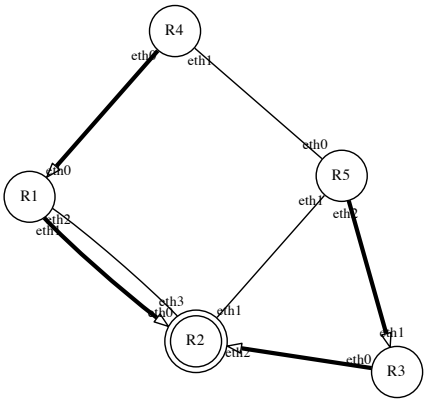
Tout non respect des consignes précédentes implique un projet non recevable et donc une note de 0



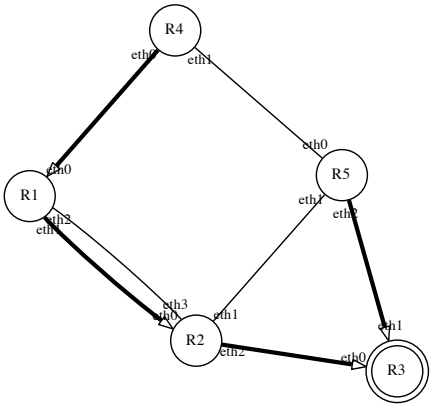
(a) Topologie complète



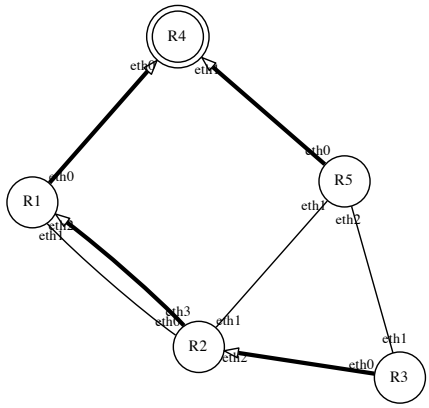
(b) Routes vers R1 (10.0.0.1)



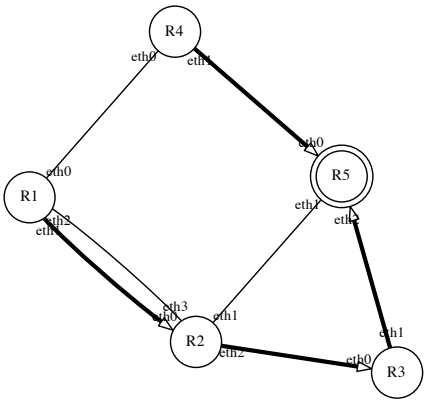
(c) Routes vers R2 (10.0.0.2)



(d) Routes vers R3 (10.0.0.3)



(e) Routes vers R4 (10.0.0.4)



(f) Routes vers R5 (10.0.0.5)

FIGURE 7 – Graphes de la topologie et des routes générés avec `NetworkGrapher` et graphviz.