



1. Introduction

1.1 Objectif

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 à vecteur de distances (*Distance Vector*).

Afin d'atteindre l'objectif du projet, il vous est demandé, d'une part, de répondre à des questions théoriques et, d'autre part, de compléter une implémentation existante d'un protocole de routage à vecteur de distances. Certains mécanismes et fonctionnalités n'ont pas encore été implémentés.

La Section 2 décrit les étapes clés qui vous sont imposées lors de la réalisation de ce projet dans le simulateur. Certaines de ces étapes clés demanderont l'implémentation de nouveaux mécanismes ainsi que l'adaptation du code existant. Des analyses vous seront également demandées. Vous devrez également résoudre sur papier plusieurs exercices portant sur la couche réseau, lors d'une séance de TP prévue à l'horaire. Cette partie écrite comptera pour 7 points sur 20. La Section 3 décrit succinctement certaines fonctionnalités du simulateur et de son interface de programmation.

1.2 Délivrables

Ce projet est à réaliser par **groupe de deux étudiants** mais la partie écrite se fait individuellement. Les étudiants n'ayant pas rendu cette feuille à la fin de la séance prévue auront une note nulle pour cette partie. L'implémentation doit être soumise sur la plate-forme Moodle pour le **vendredi 19 Mai 2023**. **Au plus tard** à cette date, à midi, vous devez avoir rendu **les livrables du projet: un rapport ainsi que le code source et une version compilée de votre implémentation**. Une fois la deadline passée plus aucun projet ne sera accepté.

Le rapport doit être **exclusivement** fourni au **format PDF**. Il contiendra les analyses et explications demandées pour chaque étape clé. Les noms, prénoms, matricules de chaque membre du groupe ainsi que le numéro du groupe doivent être mentionnés clairement en dessous du titre. Le rapport contiendra au **maximum 2 pages** de contenu (en excluant la page de titre, table des matières/figures, index et bibliographie).

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**. Le rapport et les sources (.java) de votre implémentation devront être fournis **dans une archive**. L'archive devra être nommée "tp-reseaux-2023-grX" où X est remplacé par le numéro de votre groupe. Votre archive sera fournie au format "zip" ou "tar.gz".

2. Implémentation en simulateur

La partie en simulateur est décomposée en plusieurs étapes clés qu'il vous est demandé de réaliser dans l'ordre. Pour chaque étape, les informations qui doivent absolument apparaître dans le rapport seront mentionnées.

2.1 Protocole de routage à vecteur de distances

Le package `reso.examples.dv_routing` en particulier contient une implémentation simplifiée et incomplète d'un protocole de routage à vecteur de distances. Toutes les modifications de code attendues dans ce projet auront lieu dans ce package. Dans l'archive finale qui sera soumise sur Moodle, le dossier `dv_routing` devra s'y trouver à la racine.

Le package contient une classe `Demo` dont le but est de lancer une simulation qui teste le protocole de routage à vecteur de distances. Lorsque vous lancez cette simulation, on observe que le protocole de routage a convergé et la méthode affiche pour chaque routeur l'ensemble des routes calculées. Cependant, la topologie utilisée, chargée depuis un fichier, ne contient que 2 routeurs et n'est donc pas très intéressante pour analyser le fonctionnement du protocole.

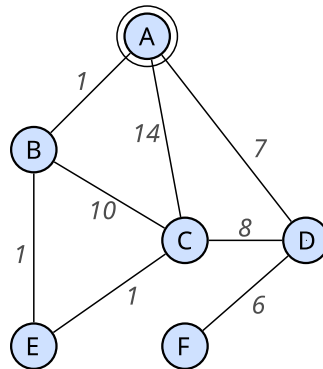


Figure 2.1: Topologie du réseau.

Il vous est demandé de définir la topologie présentée à la Figure 2.1 dans un fichier texte que vous nommerez `demo-graph.txt` (comme expliqué dans la Section 3). Enregistrez dans un fichier `demo-output.txt` les résultats générés par votre programme (i.e. l'output du terminal) pour cette nouvelle topologie.

Les fichiers `demo-graph.txt` et `demo-output.txt` devront se trouver à la racine de votre archive.

2.2 Comptage à l'infini

Une fois familiarisé avec l'implémentation du protocole de routage, il vous est demandé de mettre en évidence le problème de comptage à l'infini qui se produit lors d'un changement de métrique de lien. Pour ce faire, une nouvelle simulation devra être mise en place via l'implémentation d'une nouvelle classe nommée *Infinity* (basée sur le fonctionnement de la classe *Demo*). Elle mettra en avant le phénomène sur la topologie obtenue au point précédent (dans le fichier `demo-graph.txt`).

Cependant, lors de la réalisation de cette étape clé, vous vous rendrez compte qu'il manque une fonctionnalité importante dans l'implémentation du protocole. En effet, le protocole ne réagit pas à un éventuel changement de métrique d'un lien de la topologie. Hors, pour provoquer ce problème, il est nécessaire d'y réagir.

Adaptez donc l'implémentation du protocole afin de pouvoir mettre en place cette simulation illustrant le problème de comptage à l'infini.

La simulation configurée dans la classe *Infinity* doit également afficher les informations relatives à chaque étape de l'application du protocole et montrer que la convergence est bien plus longue qu'espérée.

Vous devez calculer les meilleures routes uniquement vers une seule destination. Enfin, enregistrez les résultats générés par votre programme dans un fichier `infinity-output.txt`. Les fichiers *Infinity.java* et `infinity-output.txt` devront également se trouver à la racine de votre archive.

2.3 Solution au problème de comptage à l'infini

Le cours théorique a présenté une solution au comptage à l'infini. Indiquez dans votre rapport le nom de cette solution et expliquez en maximum 5 lignes son principe de fonctionnement.

Implémentez cette solution dans le code source du protocole de routage. Ensuite, exécutez à nouveau votre classe *Infinity* afin de vérifier que le problème est résolu (la convergence devrait être beaucoup plus rapide). Enregistrez les nouveaux résultats générés par votre programme dans un fichier `solution-output.txt` afin de pouvoir comparer cette trace avec celle stockée dans `infinity-output.txt`. Le fichier `solution-output.txt` devra également se trouver à la racine de votre archive.

2.4 Nouveau cas exceptionnel à gérer

La solution mise en oeuvre à l'étape précédente n'est pas parfaite. En effet, elle n'empêche pas un autre problème, vu au cours, de se produire. Créez une classe *Problem* qui, similairement aux classes *Demo* et *Infinity*, permet de lancer une simulation sur une topologie pour laquelle la solution obtenue précédemment ne fonctionne pas. Cette classe doit également afficher les mêmes informations relatives à l'exécution du protocole pour le calcul des routes de chaque noeud de la topologie vers une unique destination. Enregistrez les résultats générés par votre programme dans un fichier `problem-output.txt`. La topologie utilisée pour cette expérience sera chargée à partir du fichier `problem-graph.txt` qu'il faudra créer.

Décrivez dans le rapport, en maximum 5 lignes, pourquoi le problème persiste dans ce cas exceptionnel. Proposez dans le rapport un mécanisme qui permettrait de résoudre ce nouveau problème. Vous pouvez par exemple modifier le format des messages et transporter d'autres informations qui permettraient de détecter un problème sur le chemin parcouru jusqu'à présent. Implémentez enfin ce mécanisme.

Exécutez à nouveau votre classe `Problem` afin de vérifier que ce dernier problème est résolu. Enregistrez les résultats générés par votre programme dans un fichier `solution2-output.txt`. Les fichiers `Problem.java`, `problem-graph.txt`, `problem-output.txt` et `solution2-output.txt` devront également se trouver à la racine de votre archive.

2.5 Tableau récapitulatif

Afin d'éliminer toute ambiguïté sur le contenu de l'archive à rendre sur Moodle, le Tableau 2.2 décrit la liste des fichiers qui doivent s'y trouver. Ce tableau contient également le nombre de points attribués à chaque étape du projet. La partie écrite compte pour 7 points sur 20

Etape	Fichier	Description	Points /20
2.1	<code>demo-graph.txt</code> <code>demo-output.txt</code>	topologie. trace d'exécution.	/2
2.2	<code>Infinity.java</code> <code>infinity-output.txt</code>	code simulation DV + comptage à l'infini trace d'exécution	/3
2.3	<code>solution-output.txt</code>	trace d'exécution	/2
2.4	<code>Problem.java</code> <code>problem-graph.txt</code> <code>problem-output.txt</code> <code>solution2-output.txt</code>	code simulation DV + problème topologie trace d'exécution trace d'exécution	/5
FINAL	<code>package dv_routing</code> modifié <code>rapport.pdf</code>	dossier contenant code source modifié rapport au format PDF	/1

Table 2.1: Structure de l'archive à rendre.



3. Annexes relatives au simulateur

3.1 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.

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

La syntaxe utilisée pour décrire textuellement la topologie du réseau est relativement simple. La Figure 3.1 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:

- 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.

3.2 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

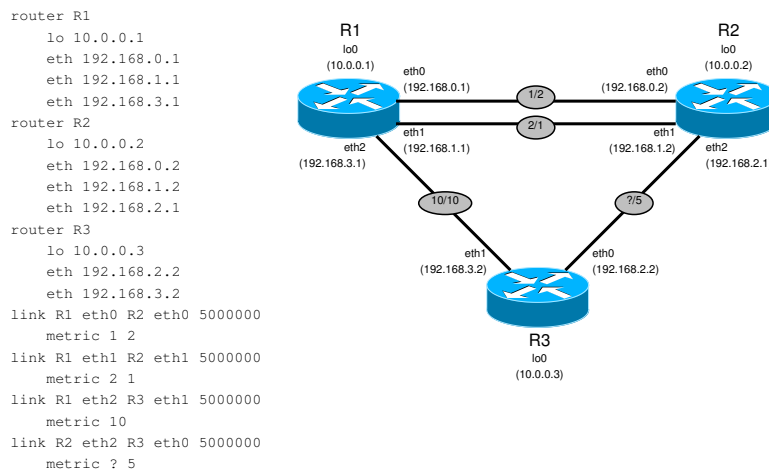


Figure 3.1: Exemple de topologie et sa représentation textuelle.

instance de l'application à ajouter.

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

```

for (Node n: network.getNodes()) {
    if (!(n instanceof IPRouter))
        continue;
    IPRouter router= (IPRouter) n;
    router.addApplication(new DVRoutingProtocol(router, true));
    router.start();
}

```

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 constructeur 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.

3.3 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*.

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

3.4 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.

```
IPInterfaceListener listener= new IPInterfaceListener() {  
    public void receive(IPInterface src, Datagram datagram) {  
        System.out.println("Datagram received: "+datagram);  
    }  
}  
IPHost ip= ...  
ip.addListener(IP_PROTO_LS, listener);
```

3.5 Lancement de la simulation

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

 **Attention!** il est possible que la méthode `run` ne se termine jamais si des événements exécutés par le simulateur ajoutent eux-mêmes de nouveaux événements 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.

3.6 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'attribut `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.

3.7 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"`).

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

```
for (Node n: network.getNodes()) {
    if (!(n instanceof IPRouter))
        continue;
```



```
IPRouter router= (IPRouter) n;  
System.out.println("Router [" + router.name + "]);  
for (IPRouteEntry re: router.getIPLayer().getRoutes())  
    System.out.println("\t" + re);  
}
```