Projet Réseaux

Simulation d'un protocole de routage à vecteur de distances

Faculté des Sciences

UMONS
Université de Mons

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 de compléter une implémentation existante d'un protocole de routage à vecteur de distances. L'implémentation est réalisée dans un simulateur qui sera décrit succintement dans ce document. Cette implémentation fournie est volontairement simplifiée et incomplète. 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. Certaines de ces étapes clés demanderont l'implémentation de nouveaux mécanismes ainsi que l'adaptation du code existant. Des analyses vous seront également demandées. La Section 3 décrit succintement le fonctionnement du simulateur et de son interface de programmation.

1.2 Délivrables

Ce projet est à réaliser par **groupe de deux étudiants**. Le projet doit être soumis sur la plate-forme Moodle pour le **vendredi 8 Janvier 2021** . **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. 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 5 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-clr-2020-gr`X" où X est remplacé par le numéro de votre groupe. Votre archive sera fournie au format "zip" ou "tar.gz".

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

2. Etapes clés

Le projet est décomposé 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 ~~Analyser et comprendre l'implémentation existante~~

Afin de vous familiariser avec le simulateur développé en Java, vous êtes fortement invités à consulter le code source des exemples d'utilisation du simulateur fournis dans le package `reso.examples`. Essayez éventuellement d'exécuter certains de ces exemples afin de mieux comprendre ce qu'ils font et les résultats qu'ils produisent. La section 3 décrit de manière détaillée le fonctionnement interne du simulateur.

2.2 ~~Protocole de routage à vecteurs de distance~~

Le package `reso.examples.dv_routing` en particulier contient une implémentation simplifiée et incomplète d'un protocole de routage à vecteurs de distance. 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 vecteurs de distance. Lorsque vous lancerez la simulation, vous constaterez que le protocole de routage affiche des informations pour chaque étape de l'application de Bellman-Ford. Lorsque la simulation est terminée (i.e. le protocole de routage a convergé), cette classe affiche pour chaque routeur l'ensemble de ses routes. 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. Il vous est demandé de définir une topologie dans un fichier texte (comme expliqué dans la Section 3) qui comporte au moins 4 noeuds et au moins 5 liens. Enregistrez dans un fichier `demo-output.txt` les résultats générés par votre programme (i.e. l'output du terminal) pour cette nouvelle topologie.

3/3

Dans le rapport, insérez une figure représentant votre topologie. Cette figure doit indiquer le coût des liens entre les routeurs. Fournissez également dans un tableau les routes calculées par le protocole à partir de chacun des routeurs. Les fichiers `demo-graph.txt` et `demo-output.txt` devront se trouver à la racine de votre archive.

2.3 ~~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.

couleurs: notes pour dépôt, tâches achevées, tâches non-achevées, commentaires.

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`). A vous de trouver une topologie dans laquelle le problème de comptage à l'infini apparaît. Cette topologie sera chargée à partir du fichier `infinity-graph.txt` que vous devez créer.

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.

8/8

La simulation configurée dans la classe `Infinity` doit également afficher les informations relatives à chaque étape de l'application de Bellman-Ford 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`, `infinity-graph.txt` et `infinity-output.txt` devront également se trouver à la racine de votre archive. Votre rapport doit contenir la topologie mise en oeuvre pour reproduire le comptage à l'infini en cas de changement de métrique. Il doit également indiquer le nombre de messages échangés par les routeurs (i.e. nombre d'itérations) depuis le changement de métrique et jusqu'à la nouvelle convergence.

De plus, sur base de cette même topologie, il vous est demandé de trouver 2 autres assignements de coûts des liens dont 1 assignement doit résulter en une convergence encore plus lente et l'autre doit résulter en une convergence plus rapide. Vous ne devez pas fournir de classes pour ces 2 simulations alternatives, il est simplement demandé d'indiquer dans le rapport, sous forme de tableau, pour chacune de ces 2 autres simulations, les nouveaux coûts attribués aux liens ainsi que le nombre d'itérations résultant pour ces attributions. Le Tableau 2.1 illustre un exemple de tableau attendu dans votre rapport pour décrire les assignations des liens pour ces 2 simulations alternatives.

Simulation	Lien R1-R2	Lien R2-R3	Lien R1-R3	Nombre d'itérations
#1	60	1	50	44
#2	?	?	?	?
#3	?	?	?	?

Table 2.1: Exemple de format attendu pour le tableau.

2.4 ~~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.

3/3

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.

2.5 Nouveau cas exceptionnel à gérer

4/7

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.

2.6 Evaluation via génération de topologies

2/3

Nous cherchons maintenant à évaluer le temps de convergence du protocole sur des topologies ayant certaines caractéristiques. Plus précisement, nous désirons voir l'impact du nombre de noeuds et de la densité d'une topologie (qui est un graphe non-orienté) sur ce temps de convergence. La densité de la topologie est définie comme la moyenne des degrés des noeuds de cette topologie, c-à-d. le nombre moyen de liens par noeud. Pour ce faire, un générateur de topologie doit être implémenté. Il doit prendre en paramètre le nombre de noeuds et la densité désirée. Le nombre de noeuds doit être égal au paramètre tandis que la densité obtenue doit se rapprocher de la valeur désirée sans forcément l'égaler.

Votre programme peut être implémenté dans le language de programmation de votre choix. De plus, la liberté vous est donnée sur la méthode à utiliser pour générer une topologie respectant les contraintes énoncées. Vous pouvez par exemple vous baser sur le mécanisme d'attachement préférentiel suivant le modèle Barabási–Albert [1]. Décrivez en maximum 5 lignes le fonctionnement de votre algorithme.

Afin de vous éviter les détails de création d'une topologie dans le générateur – tels que la gestion des adresses Ethernet et IP uniques par interface de chaque noeud –, un format simplifié de topologie (graphe non-orienté) a été défini. Votre programme devra générer un fichier respectant ce format afin qu'il puisse être lu par une nouvelle méthode de la classe `NetworkBuilder` qui se chargera d'importer la topologie.

Ce format, illustré en Figure 2.1, définit d'abord les routeurs (i.e. noeuds) et leurs positions dans le plan (en mètres). Chaque ligne représente un nouveau noeud dont l'identifiant est son numéro de ligne. Ensuite, une ligne vide permet de séparer la définition des noeuds et la définition des liens entre ces noeuds. Les commentaires (commençant par #) sont à ignorer et sont uniquement présents pour illustrer le format.

[1] `https://en.wikipedia.org/wiki/Barab%C3%A1si%E2%80%93Albert_model`

```
# Positions
0 0 # R1
30 50 # R2
50 0 # R3

# Liens (symetriques)
1 2
2 3
1 3
```

3/3

Figure 2.1: Format du fichier de topologie à générer

Une fois ce générateur implémenté, il vous est demandé de calculer le temps de convergence de 5 topologies générées aléatoirement ayant 10, 20, 30, 50 et 100 liens. La métrique utilisée pour le temps de convergence peut être le temps simulé avant l'envoi du dernier paquet du protocole ou le nombre total de paquets envoyés par le protocole.

Le rapport doit contenir un graphique (ou un tableau) contenant les temps de convergence obtenus pour chaque nombre de liens. Le code source de votre programme de génération doit également se trouver à la racine du projet. Une description de lancement du programme doit être incluse dans le rapport.

2.7 Tableau récapitulatif

Afin d'éliminer toute ambiguïté sur le contenu de l'archive à rendre sur Moodle, le Tableau 2.2 décrit la structure attendue ainsi que l'ensemble des fichiers qui doivent s'y trouver.

Etape	Fichier	Description
2.2	`demo-graph.txt` `demo-output.txt`	topologie. trace d'exécution.

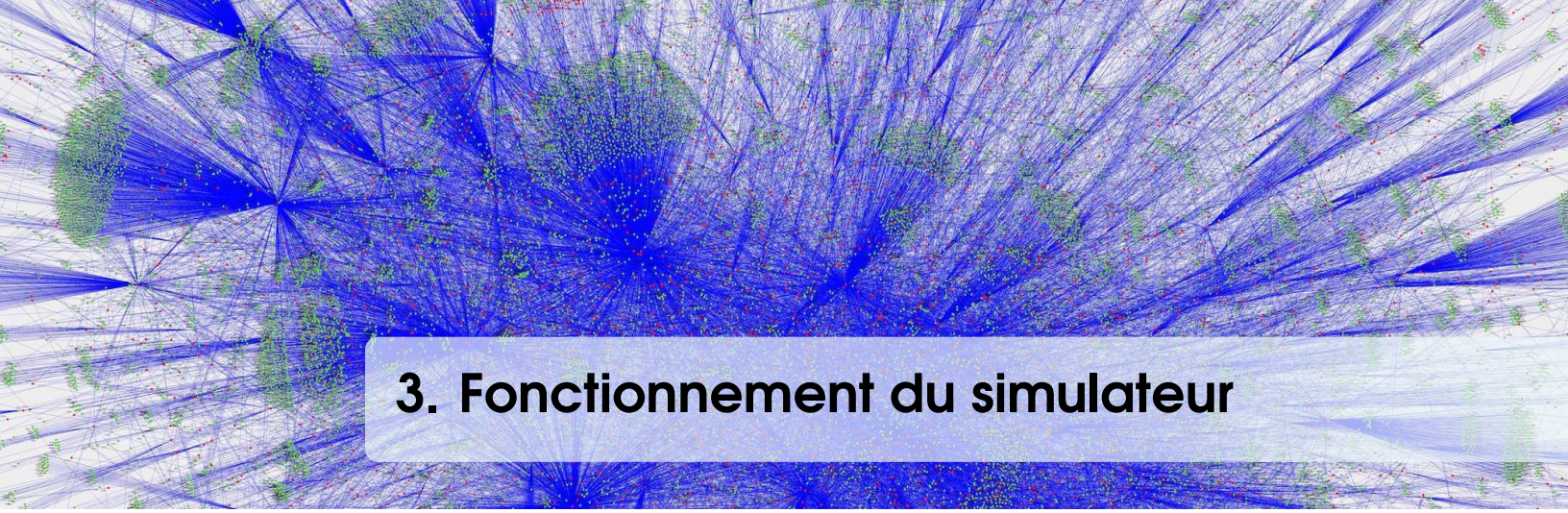
2.3	`Infinity.java`
	`infinity-graph.txt`
	`infinity-output.txt`
		code simulation DV + comptage à l'infini
		topologie
		trace d'exécution

2.4	`solution-output.txt`	trace d'exécution

2.5	`Problem.java`
	`problem-graph.txt`
	`problem-output.txt`
	`solution2-output.txt`
		code simulation DV + problème
		topologie
		trace d'exécution
		trace d'exécution

2.6	generator	dossier contenant code source du générateur
FINAL	package `dv_routing` modifié rapport.pdf	dossier contenant code source modifié rapport au format PDF

Table 2.2: Structure de l'archive à rendre.



3. Fonctionnement du simulateur

3.1 Introduction

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 3.1 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`.

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`

Représente la documentation du projet

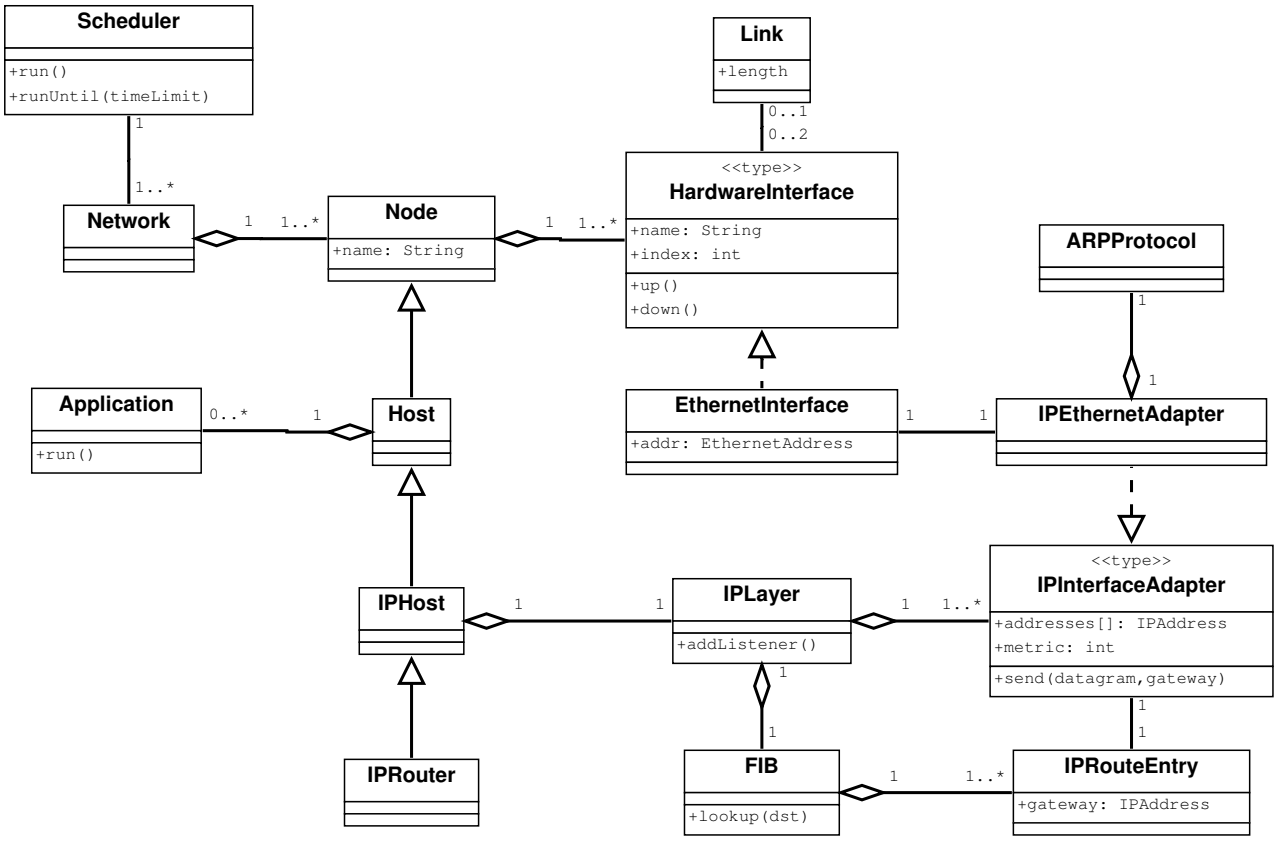


Figure 3.1: Diagramme de classes du modèle de réseaux.

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 3.2 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 data-

[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.

gramme 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.

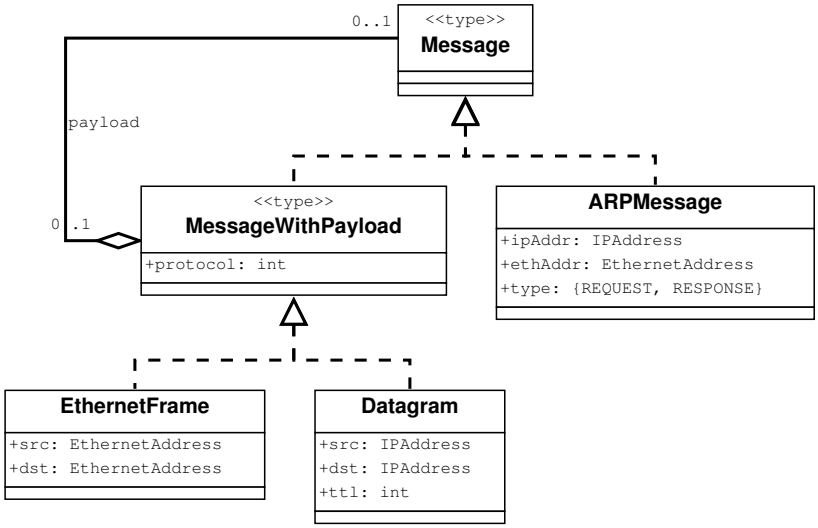


Figure 3.2: Diagramme de classes des messages.

3.2 Interface de programmation

3.2.1 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 3.3 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

[2]Le modèle actuel ne représente pas les protocoles de la couche transport.

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
```

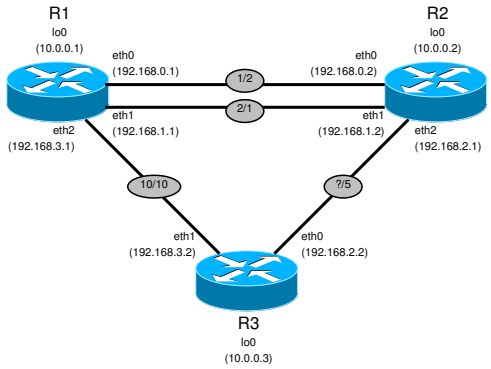


Figure 3.3: Exemple de topologie et sa représentation textuelle.

3.2.2 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.

```
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.

3.2.3 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);
```

3.2.4 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);
```

3.2.5 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.

```
private class MyTimer extends AbstractTimer {
        public MyTimer(AbstractScheduler scheduler, int interval) {
                super(scheduler, interval, true);
        }
        public void run() throws Exception {
                System.out.println("Current time: "+scheduler.getCurrentTime());
        }
}
AbstractTimer timer= new MyTimer(scheduler, 1);
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.

3.2.6 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.

3.2.7 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);
}
```

3.2.8 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.

3.2.9 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.

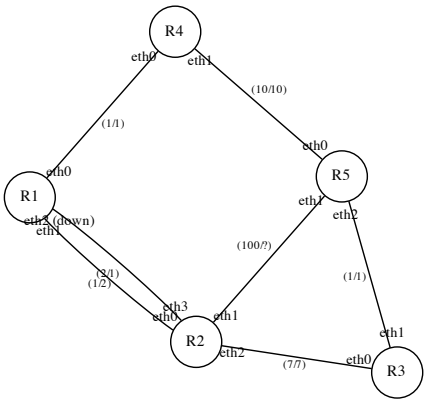
```
File f= new File("/tmp/topology.graphviz");
Writer w= new BufferedWriter(new FileWriter(f));
NetworkGrapher.toGraphviz(network, new PrintWriter(w));
w.close();
```

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 3.4(a) en utilisant la commande `neato` (partie de graphviz).

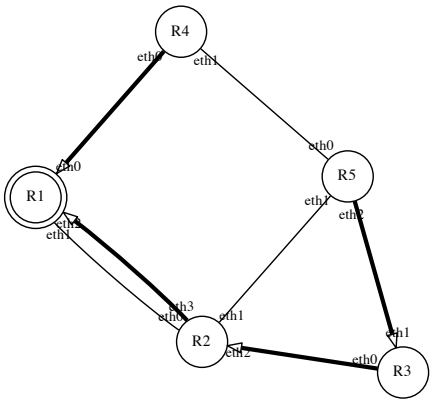
```
bash$ cat /tmp/topology.graphviz | neato -Tpdf > topology.pdf
```

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 3.4(b) à 3.4(f) montrent les routes calculées par le protocole de routage dans une simulation. 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 3.4(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`.

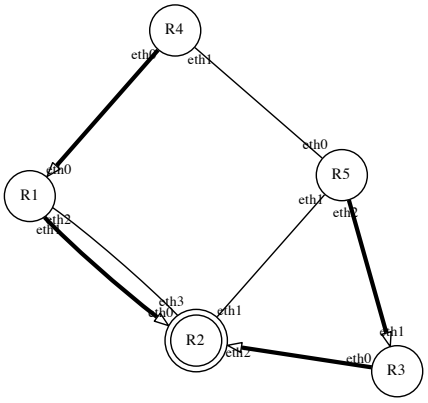
[3] L'outil graphviz est disponible gratuitement à l'adresse `http://www.graphviz.org`.



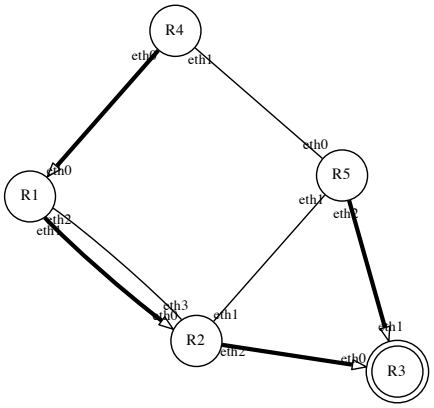
(a) Topologie complète



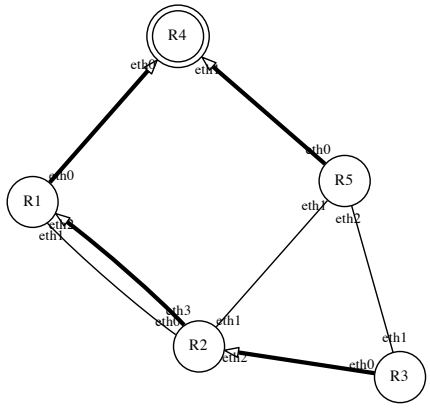
(b) Routes vers R1 (10.0.0.1)



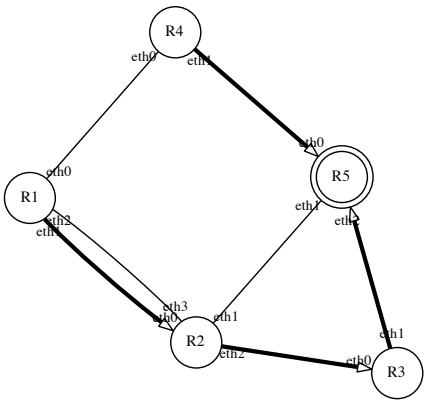
(c) Routes vers R2 (10.0.0.2)



(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 3.4: Graphes de la topologie et des routes générés avec `NetworkGrapher` et graphviz.