Réseaux I

Projet I : Go-Back-N & Congestion

Faculté
des Sciences

UMONS
Université de Mons

1. Introduction

1.1 Enoncé

Le but de ce projet est d'implémenter au sein du **bq-simulator**[1] un protocole de type go-Back-N avec un contrôle de la congestion de type *Reno*. Plus clairement vous devez créer une implémentation fonctionelle en trois parties:

1. Une partie "application" de type lambda : envoi de simples paquets et réception de l'ACK par l'émetteur. C'est ici que le numéro de séquence est décidé.
2. Une partie "go-Back-N" qui s'assure du pipelining.
3. Une partie implémentant un mécanisme de contrôle de la congestion.

1.1.1 Partie application

Cette partie gère la création d'un bête paquet (contenant un payload défini et un numéro de séquence) qui sera enoyé à la couche go-Back-N. Au niveau du destinataire un ACK sera envoyé s'il s'agit du paquet attendu (voir numéro de séquence). En fonction du fait que le paquet reçu soit correctement ordonnancé ou pas, la destination enverra un ACK cumulatif ou bien réémettra l'ack du dernier paquet correctement ordonnancé.

1.1.2 Partie go-Back-N

La partie go-Back-N doit permettre d'envoyer de mutliples paquets avec la condition qu'au pire il n'y ait pas plus de **N** paquets non acquittés. Un noeud émetteur doit répondre à trois types d'évenements:

1. Appel de la couche application
2. Réception d'un ACK
3. Timeout

1.1.3 Partie "congestion control"

Le but est d'implémenter le même mécanisme que dans *TCP reno* avec une taille de fenêtre de congestion exprimée en paquets[2]:

1. Additive increase : à chaque RTT on augmente d'une fraction d'unité (1/X) la taille de la fenêtre si on n'as pas reçu de signe de congestion.
2. Multiplicative decrease : la fenêtre de congestion est divisée par 2 à chaque perte (signalée par la réception de 3 ACKs dupliqués) avant d'employer de l'additive increase.
3. Slow start : on commence par une fenêtre de 1 slot et on augmente cette capacité de 1 slot par paquet correctement acquitté. Cette phase se termine au delà d'un certain seuil où on switchera alors pour l'additive increase.

[1] Les sources du simulateur sont disponibles sur moodle.

[2] En effet pour ce projet on suppose que chaque paquet à la même taille

4. Réaction à un timeout : retour à une taille de 1 + slow start.

1.2 Remarques générales

- Toutes les infos nécessaires sur go-Back-N et le contrôle de la congestion sont dans votre cours.
- Un plot doit être fourni pour pouvoir suivre l'évolution de la taille de la fenêtre!!!
- Pensez à un système de log efficace, pour pouvoir observer l'évolution de la taille de la fenêtre de congestion, ainsi que les événements (perte d'un paquet, timeout,...)
- Le taux de transmission de paquets doit pouvoir être paramétrable
- Au niveau du receveur, vous pouvez/devez rajouter du "random" pour simuler la perte d'un paquet ou un timeout.
- Créez un nouveau package intitulé *reso.examples.gobackn*
- Inspirez vous de *reso.examples.pingpong* pour créer votre application
- Pensez au scheduler pour l'envoi de paquets
- Vous n'avez pas à vous soucier outre-mesure du routage où de ce qui se passe plus bas dans la pile réseau
- Pensez à comment réduire la taille de la fenêtre s'il reste des paquets dedans (l'idée est de ne pas les perdre)

1.3 Informations pratiques

Avant même de commencer le projet, il est primordial de comprendre les différentes notions qui seront utilisées :

1. Comprendre l'une des problématiques du projet : le pipelining (voir Cours Théorique, Chap. 3, page 48-54).
2. Comprendre le fonctionnement théorique de go-Back-N (voir Cours Théorique, Chap. 3, page 55-61). Qu'as-t'on besoin de garder en mémoire pour faire fonctionner le protocole ? Quelles variables nous intéressent ?
3. Comprendre la seconde problématique du projet : le contrôle de la congestion (voir Cours Théorique, Chap. 3, page 126-131).
4. Comprendre le fonctionnement théorique de TCP Reno (voir Cours Théorique, Chap. 3, page 135).

Une fois que ces notions sont assimilées, vous pouvez vous intéresser au simulateur BQ-simulator (disponible sur Moodle). Pour cela vous pouvez :

1. lire la section "Fonctionnement du Simulateur" présente dans l'énoncé du projet.
2. Tester l'exemple PingPong. Pour cela, il suffit de lancer la classe Demo présente dans le dossier "src/reso/examples/pingpong"

1.4 Délivrables

Ce projet est à réaliser par **groupe de deux étudiants**. Le projet doit être terminé pour le **11 mai 2018** . Au plus tard à cette date, 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 poster sur moodle.

Veuillez suivre scrupuleusement les consignes indiquées ci-dessous.

Le rapport doit être 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-2018-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-2018-grX/
  rapport.pdf
  src/
    ensemble des fichiers source (.java)
  build/
    ensemble des fichiers compilés (.class)
```

Notez également que nous accordons une certaine importance à la présence de javadoc et de documentation du code

2. Utilisation du Simulateur

2.1 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 2.1 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` qui

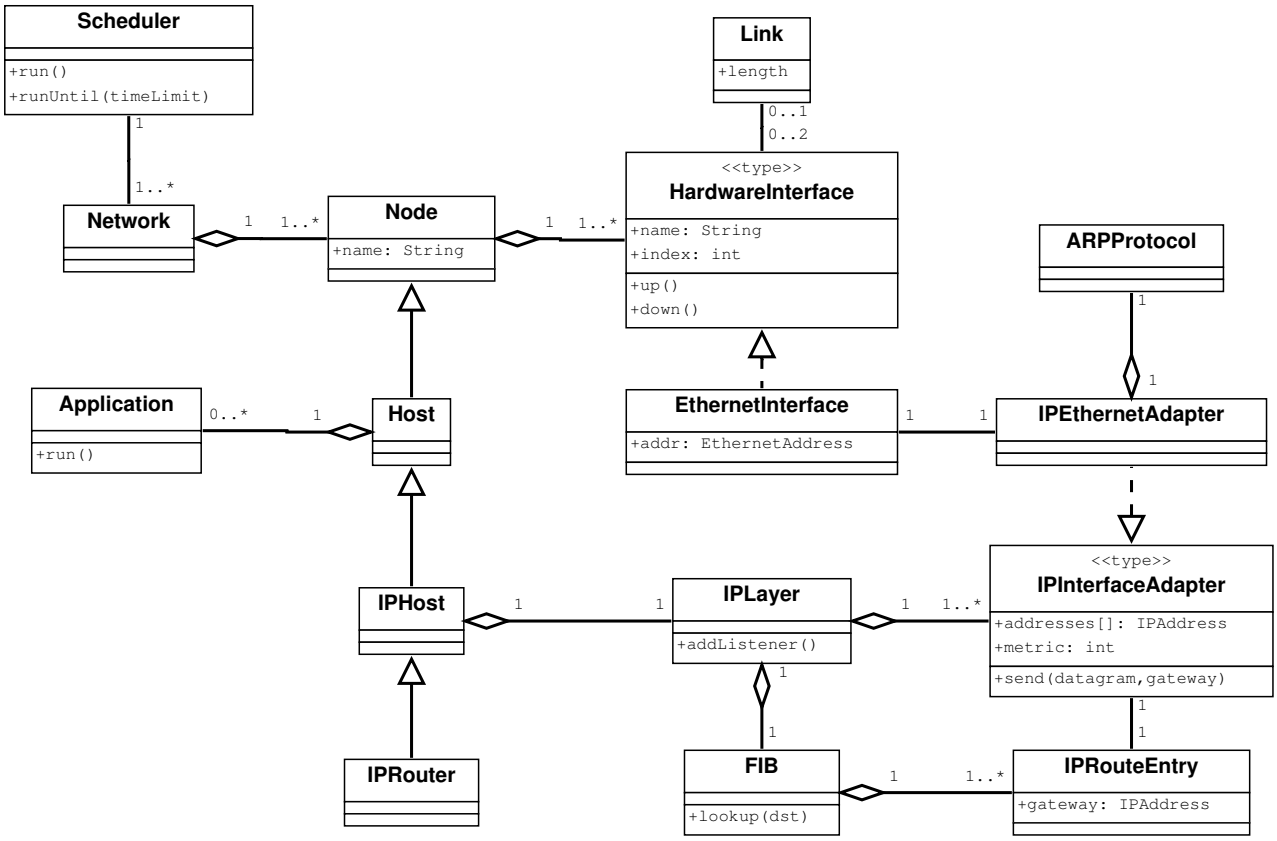

Figure 2.1: Diagramme de classes du modèle de réseaux.

fait le lien entre IP et une interface Ethernet.

La Figure 2.2 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 datagramme IP. Cette dernière peut transporter des messages de différents types (typiquement de la couche transport[1]). La classe `ARPMessage` représente les messages utilisés par le protocole ARP.

2.1.1 Interface de programmation

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

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

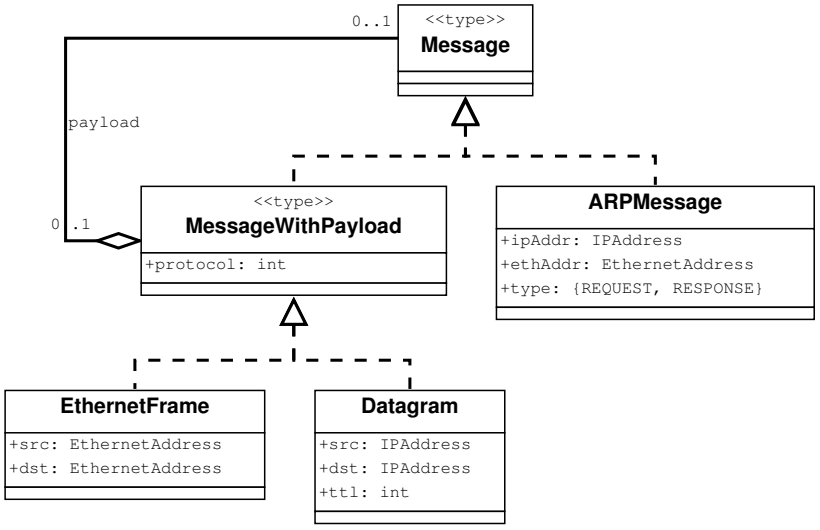

Figure 2.2: Diagramme de classes des messages.

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 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.

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);
```

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);
```

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();
```