Développement d'un protocole de routage

1 Présentation générale du sujet

L'objectif de ce travail est de développer un protocole de routage basé sur l'algorithme de Bellman-Ford, et inspiré du protocole RIP v2 (*Routing Information Protocol*) vu en Cours/TD. Pour cela vous allez vous appuyer sur un environnement logiciel que l'on vous fournit et que vous devrez compléter.

Pour rappel, un réseau comprend traditionnellement deux plans :

— le *plan de données* qui regroupe les protocoles et algorithmes utilisés par les hôtes et les routeurs pour créer et manipuler des paquets contenant les données des utilisateurs ;

— le *plan de contrôle* qui regroupe les protocoles et algorithmes utilisés pour construire les tables de routage présentes dans les routeurs du réseau.

Le plan de contrôle le plus simple est de configurer manuellement les tables de routage de tous les routeurs du réseau. Cette solution n'est pas viable pour d'une part, des réseaux de grande taille et d'autre part, la prise en compte des erreurs ou pannes pouvant survenir sur des liens ou routeurs. Il existe plusieurs approches pour déterminer « automatiquement » les routes entre les différents nœuds du réseau. Parmi ces techniques, se trouvent les protocoles de routage s'appuyant des algorithmes distribués. Dans ces protocoles, deux grandes familles existent : 1) routage par *vecteur de distances* (ou *vecteur distance*) et 2) routage par *état de lien*. Nous nous intéresserons dans la suite de ce travail au **routage par vecteur de distances**.

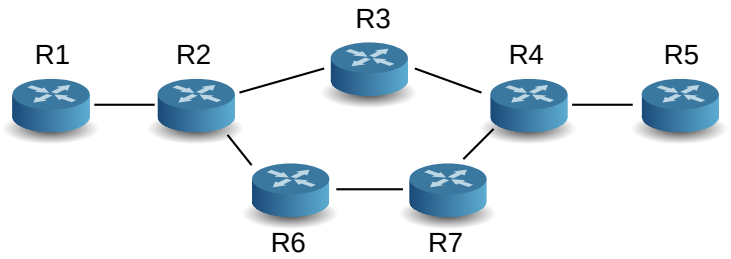

FIGURE 1 – Exemple de topologie impliquant 7 routeurs

Dans un réseau IP, chaque routeur maintient une table de routage qui stocke, en particulier, pour chaque destination, le prochain saut pour atteindre cette destination, le coût et l'ancienneté de la route. Par exemple, le tableau 1 donne la table de routage du routeur `R2` de la topologie de la figure 1.

TABLE 1 – Table de routage du routeur `R2`

Destination	NextHop	Metric	Time
R2	-	0	20sec
R1	Addr(R1)	1	15sec
R3	Addr(R3)	1	15sec
R6	Addr(R6)	1	15sec
R4	Addr(R3)	2	15sec
R7	Addr(R6)	2	15sec
R5	Addr(R3)	3	15sec

Un routeur qui utilise un protocole de routage par vecteur de distances envoie périodiquement à ses voisins un vecteur distance qui est un sous-ensemble de sa table de routage et qui contient la

distance pour chaque destination connue. La distance représente le coût de la route, c'est-à-dire, dans notre cas, le nombre de sauts pour atteindre la destination. Lors du démarrage du routeur, sa table de routage ne contient qu'une seule entrée : lui-même avec un coût de 0. Il diffuse le vecteur correspondant sur chacune de ses liaisons. Lorsqu'un routeur reçoit un vecteur distance, il applique l'algorithme vu en Cours/TD et rappelé plus loin dans le sujet. Cela permet au final de déterminer le chemin le plus court pour atteindre une destination donnée.

2 Environnement de travail

Dans l'environnement que vous allez utiliser, chaque routeur est considéré comme un processus à part entière. Le réseau est constitué donc d'un ensemble de processus qui communiquent via l'API des sockets TCP/IP. Le résultat est un réseau dit « superposé » au réseau réel, on parle alors de *overlay network*.

2.1 Architecture logicielle du routeur

L'architecture logicielle du routeur que vous allez implémenter est schématisée sur la figure 2. Le routeur est composé d'un programme principal (*main*) qui démarre deux fils d'exécution (*thread*) :

1. `Process input packets` qui se charge d'écouter les paquets entrants et de leur appliquer le traitement adapté (relayage, mise à jour table de routage...).
2. `Hello` qui a pour rôle de diffuser périodiquement aux voisins des informations de contrôle, en l'occurence ici le vecteur distance du routeur.

Les interactions avec l'utilisateur se font dans le module `console` : affichage de la table des voisins, table de routage, commandes de tests, etc.

Dans la suite, l'essentiel du travail demandé s'insère dans les deux *threads* `Process input packets` et `Hello`. Les autres parties ont déjà été développées.

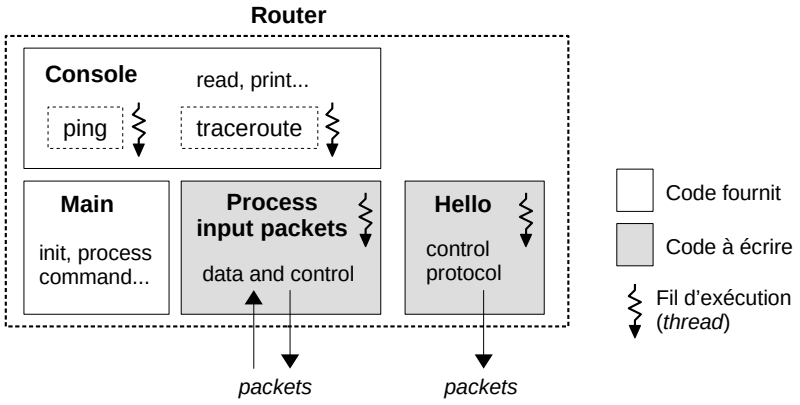

FIGURE 2 – Architecture logicielle du routeur

2.2 Structure des fichiers et test d'une topologie

Tout le code source du programme se trouve à la racine du dossier principal. Les journaux d'événements des routeurs se trouvent dans le dossier `log/` (messages générés à partir des fonctions de log), et les topologies de test dans le dossier `topo/`. Cinq topologies sont prédéfinies (`t1.txt` à `t5.txt`), vous pouvez en rajouter de nouvelles si vous le souhaitez.

Nous vous fournissons un `Makefile` pour faciliter la compilation et l'exécution des différents routeurs de la topologie. Pour générer l'exécutable `router`, il vous suffit de saisir `make router`. Pour exécuter une topologie complète : `make test_topoX` avec `X` l'identifiant de la topologie. Exemple pour démarrer la topologie t4 :

```
$ make clean
$ make test_topo4
```

Une fois que vous avez démarré une topologie, un terminal s'ouvre pour chaque routeur. La liste des commandes est accessible sur un routeur via la commande `help`. Exemple :

```
R1> help
Commands:
  clear               Clear the terminal screen.
  ping <id>           Send echo request to node <id>.
  pingforce <id>      Send echo request until response or timeout (1min).
  show ip neigh       Show neighbors table.
  show ip route       Show IP routing table.
  traceroute <id>     Print the path to destination <id>.
  help                Show help for commands.
```

En particulier, la commande `ping` vous permet de faire un test de connectivité et `traceroute` un test de tracé de route. Exemple sur la topologie 4 de la figure 1 :

```
R1> traceroute 5
Traceroute to R5, 64 hops max.
  1       R2       0.025s
  2       R3       0.047s
  3       R4       0.068s
  4       R5       0.093s
R1>
```

La destruction de toute la topologie se fait via `make kill_test`.

3 Structures de données manipulées

3.1 Identifiant et adresse d'un nœud (router.h)

Chaque routeur du réseau est identifié de manière unique via un petit entier positif (0 à 255) de type `node_id_t`. De plus, pour pouvoir localiser le routeur dans le réseau superposé, un autre synonyme de type est utilisé qui inclut aussi l'adresse IPv4 et le numéro de port UDP.

```
// Small unsigned integer as node ID
typedef unsigned char node_id_t;

// Overlay address
typedef struct {
    node_id_t id;
    char ipv4[IPV4_ADR_STRLEN];
    unsigned short int port;
} overlay_addr_t;
```

3.2 Table de routage (router.h)

Chaque routeur dispose d'une table de routage dont chaque entrée contient : l'identifiant du routeur destinataire (`dest`), l'adresse du prochain saut pour atteindre la destination (`nexthop`), le coût en nombre de sauts (`metric`) et l'ancienneté de l'entrée (`time`), c.-à-d. la date à laquelle l'entrée a été saisie ou mise à jour.

```
typedef struct {
  node_id_t      dest;
  overlay_addr_t nexthop;
  unsigned char  metric;
  time_t         time;
} routing_table_entry_t;
```

```
typedef struct {
  unsigned short int    size;
  routing_table_entry_t tab[MAX_ROUTES];
} routing_table_t;
```

3.3 Paquets de données et de contrôle (packet.h)

Vous allez manipuler deux types de paquet : `packet_ctrl_t` qui représente un paquet de contrôle utilisé par le protocole de routage, et `packet_data_t` qui représente un paquet de données utilisé par les « applications », à savoir ici `ping` et `traceroute`. Lorqu'un routeur reçoit un paquet, il utilisera le premier octet (champ `type`) pour démultiplexer vers le bon format.

```
// Distance vector entry
typedef struct {
    node_id_t     dest;
    unsigned char metric;
} dv_entry_t;

// Control packet
typedef struct {
    unsigned char type; // CTRL
    node_id_t     src_id;
    unsigned char dv_size;
    dv_entry_t    dv[MAX_DV_SIZE];
} packet_ctrl_t;
```

```
// Data packet
typedef struct {
    unsigned char type; // DATA
    unsigned char subtype;
    node_id_t     src_id;
    node_id_t     dst_id;
    unsigned char ttl;
    unsigned char msg_seq;
    unsigned long time_sec;
    unsigned long time_nsec;
} packet_data_t;
```

4 Travail à réaliser

Le travail à réaliser comporte plusieurs étapes. Le barème indicatif de chaque partie est donné à la fin du sujet. **Le développement doit se faire dans l'ordre indiqué, chaque partie doit être terminée avant de passer à la suivante. Le développement doit se faire en langage C.**

4.1 Fonction de relayage (*forwarding*)

Cette première étape consiste à implémenter la fonction de relayage d'un paquet, c'est-à-dire la fonction qui va consulter la table de routage pour transférer le paquet de données vers le prochain saut. Cette fonction a pour en-tête :

```
int forward_packet(packet_data_t *packet, int psize, routing_table_t *rt)
```

avec `packet` qui désigne un pointeur vers le paquet à relayer de taille `psize`, et `rt` un pointeur vers la table de routage devant être consultée pour prendre la décision. La fonction renverra 1 si le relayage s'est bien passé, 0 dans le cas où aucune route n'a été trouvée pour la destination. Le transfert du paquet se fera en utilisant l'API des sockets en mode datagramme (client utilisant le protocole UDP).

Dans un deuxième temps, cette fonction devra être appelée dans le fil d'exécution `process input packets` dans le cas où le routeur n'est pas le destinataire du paquet de données. Le traitement devra consister à décrémenter le champ TTL du paquet, si ce dernier arrive à 0 alors

renvoyer un paquet « temps écoulé » via la fonction `send_time_exceeded()` déjà écrite, sinon relayer le paquet via votre fonction `forward_packet()`.

Pour tester cette première étape, vous utiliserez la cible `test_forwarding` du `Makefile` qui lance la topologie spécifiée dans le fichier `t3.txt` et dans laquelle les tables de routage sont déjà pré-remplies :

```
$ make test_forwarding
```

4.2 Diffusion du vecteur de distances (v1)

Vous développerez ici une première version du fil d'exécution `hello` qui se charge de construire le vecteur distance et de le diffuser à tous les voisins.

La construction du paquet de contrôle contenant le vecteur distance se fera avec la fonction d'en-tête :

```
void build_dv_packet(packet_ctrl_t *packet, routing_table_t *rt)
```

avec `packet` qui désigne un pointeur vers le paquet de contrôle à construire à partir des informations présentes dans la table de routage pointée par `rt`.

Une fois que vous avez construit le paquet de contrôle, vous devez le diffuser à tous les voisins du routeur. Pour cela, vous utiliserez l'API des sockets en mode datagramme (client utilisant le protocole UDP, la partie serveur ayant déjà été développée dans le fil `process input packets`).

Pour tester cette partie, vous utiliserez la fonction `log_dv()` que vous appellerez une fois le paquet envoyé et qui permet de journaliser dans un fichier `log/Ri.txt` cet envoi ainsi que le contenu du vecteur distance. Exemple :

```
$ make clean
$ make test_topo1
```

Puis ensuite, vérifiez dans `log/R2.txt` que le vecteur distance de R2 est bien diffusé à R1 et à R3.

4.3 Mise à jour de la table de routage (v1)

Cette partie consiste à développer la mise à jour de la table de routage en fonction du vecteur distance reçu. Ce traitement se fera dans la fonction d'en-tête :

```
int update_rt(
       routing_table_t *rt, overlay_addr_t *src, dv_entry_t dv[], int dvsize)
```

où `rt` désigne un pointeur vers la table de routage, `src` un pointeur vers l'adresse source du routeur émettant le vecteur distance représenté par le tableau `dv`, de taille `dvsize`. Cette fonction devra implémenter le pseudo-code suivant :

```
# SRC : routeur source
# DV : vecteur reçu de SRC
# RT : table de routage
Pour chaque entrée e de DV faire
    si (e.dest n'existe pas dans RT) alors
      ajouter une nouvelle route vers e.dest
      avec un cout de e.metric+1
    sinon # route existante = r
      si (cout(r)>e.metric+1 ou prochain_saut(r)=SRC) alors
        mettre à jour r
```

Cette fonction devra être appelée dans le fil d'exécution `process input packets` lorsque le paquet reçu est un paquet de contrôle.

Vous testerez votre programme sur les topologies t1 à t5. En particulier, vous vérifierez que le chemin le plus court est bien utilisé lors de l'utilisation de l'outil `traceroute` (cf. section 2.2). Vous vérifierez également que l'ancienneté de la route est bien mise à jour lorsque vous affichez la table de routage (colonne `LifeTime` dans `show ip route`).

4.4 Gestion de la perte de liens et de routeurs

La version actuelle de votre programme ne prend pas en charge les changements de topologie, comme la perte de liens ou de routeurs. Par exemple, lorsque vous démarrez la topologie 1, puis que vous arrêtez le routeur R2, ce dernier reste toujours présent dans la table de routage de R1 et R3, entrainant une perte de tous les paquets routés via R2.

4.4.1 Table de routage (v2) : suppression des entrées obsolètes

Une solution classique au problème décrit précédemment est de prendre en compte la durée de vie des entrées dans la table de routage. Lorsqu'une entrée atteint une durée maximale sans avoir été rafraîchie, alors elle est supprimée de la table. Ce traitement sera codé dans la fonction d'en-tête :

```
void remove_obsolete_entries(routing_table_t *rt)
```

qui est elle-même appelée périodiquement dans le fil d'exécution `hello`. Notez que la fonction `difftime()` de la bibliothèque `time.h` pourra être utilisée pour calculer la durée de vie d'une entrée. La valeur maximale de cette durée peut être égale à la période de diffusion du vecteur distance (constante `BROADCAST_PERIOD`). Attention, dans tous les cas il faudra veiller à ne pas retirer l'entrée dont la destination est le routeur lui-même.

Vous pourrez par exemple tester cette évolution sur la topologie t1. Après avoir démarré cette topologie, arrêtez le routeur R2 et vérifiez bien que sur R1 il n'y a plus de route vers R2 et R3 (message « `no route to destination` » après un `ping`).

4.4.2 Problème du *count to infinity*

La suppression des entrées obsolètes ne résout pas tous les problèmes dans la mise à jour des tables de routage à partir des vecteurs distance reçus. Nous allons en illustrer un sur la topologie t3 de la figure ci-dessous.

Si R3 tombe en panne, R1 cessera de recevoir le vecteur distance de R3 et va donc retirer notamment la route vers R3 qui a un coût de 1. Mais le vecteur distance reçu de R4 annonce R3 à un coût de 2 ! R1 va donc rajouter une entrée dans sa table vers R3 avec une métrique de 3 via R4. Lorsque R4 supprimera sa route vers R3 (pour cause de durée de vie obsolète), il réapprendra la route par R1 avec un coût de 4... et ainsi de suite. Ce problème, désigné sous l'appellation *count to infinity*, est bien connu d'un routage à vecteur de distances.

Nous vous encourageons à tester ce scénario pour reproduire le problème, bien visible si vous faites un `traceroute` vers R3 à partir de R1.

a) Le problème du *count to infinity* se produit ici car le routeur R1 annonce une route à R4 qu'il a apprise via le routeur R4 lui-même. Une solution possible pour éviter ce problème est de changer la façon dont un routeur crée son vecteur distance. Plutôt que de construire un vecteur unique et de le diffuser à tous ses voisins, il pourrait construire un vecteur spécifique à chaque voisin qui ne contiendrait que les routes qui n'ont pas été apprises par ce voisin (technique dite du *split-horizon*). C'est le travail que nous vous demandons de faire dans la fonction d'en-tête :

```
void build_dv_specific(packet_ctrl_t *p, routing_table_t *rt, node_id_t neigh)
```

qui construit le paquet de contrôle pointé par `p` à destination du voisin `neigh`, à partir de la table de routage pointée par `rt`.

Il vous faudra ensuite mettre à jour le fil d'exécution `hello` pour appeler cette nouvelle fonction à la place de la première version écrite dans la section 4.2.

Testez cette version sur les différentes topologies en vous assurant que les chemins les plus courts sont recalculés correctement en cas d'arrêt d'un routeur (exemple : arrêt du routeur R3 sur la topologie t4).

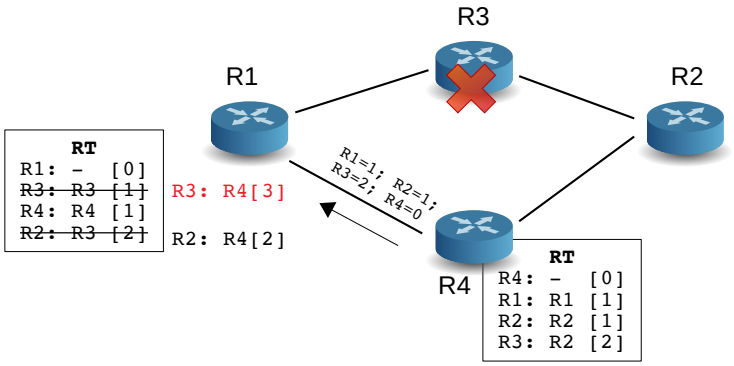

FIGURE 3 – Panne de R3 dans la topologie *t3*

b) Malgré l'amélioration précédente, il se peut qu'il y ait encore le problème du *count to infinity* sur certaines topologies comprenant des boucles. C'est le cas notamment sur la topologie t5 que l'on vous fournit. Par exemple, si le routeur R1 s'arrête, le routeur R4 va continuer de diffuser pendant un petit moment à R3 une route pour joindre R1 via R5 (ou inversement). Cette route va circuler jusqu'à R2 puis revenir vers R5 par la boucle... La technique précédente ne permet pas de détecter ce cycle.

Pour répondre à ce problème, certains protocoles imposent que la métrique ne dépasse pas une valeur maximale (ce qui limite le « diamètre » du réseau), par exemple 16 dans le cas de RIPv2. Modifiez votre programme (à vous de définir où) pour intégrer cette nouvelle fonctionnalité et testez le résultat sur la topologie t5.

5 Evaluation

Ce mini-projet est à réaliser individuellement. Vous disposez de 4 séances de TP, plus du travail personnel entre ces séances. Vous devrez déposer sur moodle une archive de votre code source commenté, au plus tard le jour de votre dernière séance de TP.

A titre indicatif, le barème sera le suivant :

— Fonction de relayage : 3 pts (section 4.1)
— Construction et diffusion du vecteur distance v1 : 4 pts (section 4.2)
— Mise à jour table de routage v1 : 3 pts (section 4.3)
— Mise à jour table de routage v2 : 2 pts (section 4.4.1)
— Problème du *count to infinity* : 4 pts (section 4.4.2)
— Suivi : 2 pts (avancement au cours des séances)
— Code : 2 pts (lisibilité, commentaires, etc.)