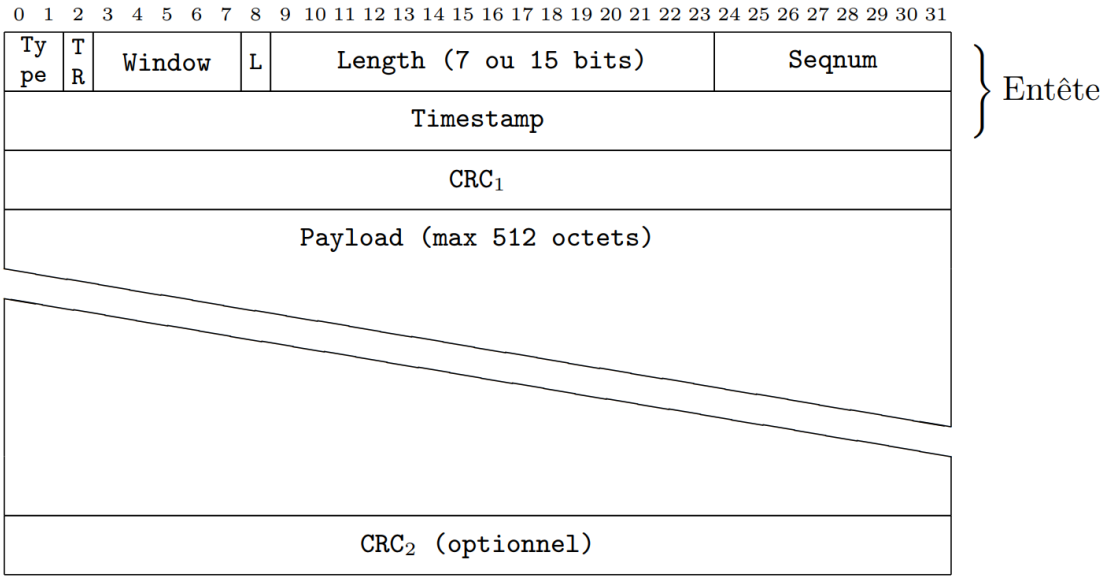
ÉCOLE POLYTECHNIQUE DE LOUVAIN

UCLouvain

LINGI 1341 - Réseaux informatiques - 2019

TRTP : Truncated Reliable Transport Protocol



Groupe 118

Camberlin	Merlin	0944-17-00
Pisvin	Arthur	3711-17-00

Remis le : 6 mai 2020

Professeur :
Bonaventure O.

Assistants :
Michel F.
Piraux M.

1 Fonctionnement général

Le fonctionnement général du `receiver` peut être décrit en plusieurs étapes :

Dans un premier temps, à partir de l'adresse `IPv6` (ou du nom de domaine) et du numéro de port fournis, la chaîne de caractères reçue est convertie en une structure de données (`sockaddr_in6`) utilisable par le système d'exploitation. Cette étape est réalisée par la fonction `real_address()` se trouvant dans le fichier `socket.c`.

Dans un second temps, un `socket` est créé et lié à la source des connexions à accepter à l'aide de la structure de données précédemment créée avec `real_address()`. La création du `socket` permet l'écoute et la réception des messages entrants. C'est la fonction `create_packet()` se trouvant dans le fichier `socket.c` qui est responsable de cette étape.

Dans un troisième temps, la fonction `read_write_loop()` lit le contenu du socket et l'écrit dans un fichier, tout en permettant de renvoyer des accusés de réception aux émetteurs sur ce même `socket`. La lecture et l'écriture est traitée simultanément à l'aide de l'appel système : `poll()`. Dès que de la donnée à lire sur le `socket` est détectée, la fonction `read_write_loop()` va récupérer l'adresse de l'émetteur du segment de données. Si l'émetteur envoie son premier message et que le nombre de connexions actuellement en cours est inférieur au nombre de connexions maximal autorisé, alors le segment de données est considéré et le traitement du segment de données peut continuer. Dans le cas contraire, le segment de données est simplement ignoré.

Dans un quatrième temps, le segment de données va être converti en une structure de données `pkt` à l'aide de la fonction `decode()` se trouvant dans le fichier `packet_implem.c`. Une fois converti, le paquet est ajouté dans la liste chaînée associée à l'émetteur du paquet si le paquet reçu n'est pas en séquence. Au contraire, si le paquet a un numéro de séquence correspondant au numéro de séquence attendu, le contenu du paquet est directement écrit dans le fichier associé à l'émetteur. Un accusé de réception est alors renvoyé pour confirmer la bonne réception du paquet. C'est toujours `read_write_loop()` dans `read_write_loop.c` qui est responsable de cette étape.

Enfin, le processus recommence à l'étape 3, jusqu'à ce que le dernier paquet d'un émetteur soit détecté. Dans tel cas, la connexion qui lui était associée est alors fermée.

2 Choix d'implémentation

2.1 Structures de données utilisées

Structure `pkt` La structure `pkt` sert à interpréter les données reçues par le `sender` dans une structure de données utilisable par le système d'exploitation.

```
struct __attribute__((__packed__)) pkt
{
  uint8_t window:5;
  uint8_t tr:1;
  uint8_t type:2; //le premier byte de notre structure represente les trois champs window, tr et type
  uint16_t length; // le champs length est encode sur 16 bits mais c'est un varuint
  uint8_t seqnum;
  uint32_t timestamp;
  uint32_t crc1;
  char* payload ; //le payload est represente par un pointeur vers une zone memoire qui contient le payload
  uint32_t crc2;
};

```

Structure Connexion Cette structure sert à gérer l'état des différentes connexions concurrentes. Cette structure possède 7 champs :

```
typedef struct connexion
{
  struct sockaddr_in6 client_addr; //l'adresse du client qui nous contacte
  int fd_to_write; // File descriptor pour ecrire dans le fichier correspondant a cette connexion
  int closed; // Ce champ designe l'etat de la connexion. Par defaut les connexions ne sont pas terminee = 0
  uint8_t tailleWindow; // taille de notre fenetre de reception
  int windowMin ; // entier compris entre 0 et 255 designant la limite basse de notre fenetre de reception
  int windowMax; //  entier compris entre 0 et 255 designant la limite haute de notre fenetre de reception
  node_t** head; //ce dernier champ designe le debut la liste chainee LinkedList. Celle-ci sera detaillee par la suite.
}connexion;

```

Structure `Node` Cette structure compose `LinkedList` . Celle-ci sert de buffer pour sauver les structures `pkt` dans l'ordre des numéros de séquence. Nous avons choisi ce mode de tri pour pouvoir écrire le contenu des paquets dans l'ordre dans le fichier. La structure Node est composée de 3 champs :

```
typedef struct node
{
  pkt_t* pkt; //la structure pkt representant le paquet
  int indice; //un entier designant sa place dans la liste. Il correspond au numero de sequence de paquet pkt.
  struct node* next; //un pointeur vers la prochaine structure Node dans la liste chainee.
}node;

```

Structure `c_node` Cette structure compose `ConnexionList`. Cette autre liste chaînée sert à gérer les connexions des différents clients. Cette structure se compose de 2 champs :

```
typedef struct c_node
{
  connexion_t* c_connexion; //la structure connexion lie a la connexion traitee
  struct c_node* next; // un pointeur vers le noeud suivant
}c_node;

```

2.2 Technique pour traiter les connexions concurrentes

La lecture et l'écriture sur le `socket` est traitée simultanément à l'aide de l'appel système `poll()`.

Comme introduit précédemment dans la description générale du programme, dès que de la donnée à lire est détectée, l'adresse de l'émetteur est récupérée pour déterminer si l'émetteur n'en est pas à son premier message envoyé.

Pour le vérifier, la fonction `contains()` dans `ConnexionList.c` parcourt la liste chaînée des connexions actuellement en cours et vérifie que l'adresse du client n'est pas déjà connue. Si en effet, l'émetteur n'est pas dans la liste chainée et que le nombre de connexions en cours est inférieur au nombre de connexions maximum, il est ajouté en tête de liste par la fonction `add()` se trouvant dans `ConnexionList.c`.

Au contraire, si il existe déjà, la fonction `contains()` retourne un pointeur vers cette connexion. A l'aide de ce pointeur, le traitement du paquet peut continuer pour notamment écrire sur le `file descriptor` qui lui est associé ou l'ajouter dans sa liste chaînée de paquets.

2.3 Mécanisme de maintient des différentes connexions concurrentes

Comme indiqué dans la section 2.1, chaque connexion a sa propre structure qui contient une variable `closed` qui indique l'état de la connexion : fermée ou pas encore. L'état est dit fermé (`closed = 1`) lorsque la transmission est terminée et est dit ouvert dans le cas contraire.

2.4 Mécanisme de fermeture de connexion

A la réception du paquet marquant la fin de la transmission, la variable `closed` est mise à 1 pour indiquer que la connexion peut être fermée, que le `file descriptor` d'écriture des paquets peut également être fermé et que la connexion peut être retirée de la liste chaînée des connexions actuellement en cours.

2.5 Mécanisme de fenêtre de réception

Pour la fenêtre de réception, nous avons choisi de regarder la longueur du champ `window` du paquet reçu et de changer la longueur de notre fenêtre en fonction de cela. En effet, nous avons les champs `windowMin`, `windowMax` et `tailleWindow` qui nous indique respectivement la limite basse, la limite haute et la taille de la fenêtre de réception spécifique à chaque connexion. Nous prenons en compte seulement les paquets ayant des numéros de séquence compris entre `windowMin` et `windowMax`. Les autres sont ignorés. Ces deux champs étant entre 0 et 255, nous avons implémentè notre code pour qu'il prenne en compte le fait que la limite haute puisse être plus petite que la limite basse. Par exemple, si notre limite basse vaut 254 et la limite haute vaut 1, nous attendons les numéros de séquence 254, 255, 0 et 1.

2.6 Stratégie pour la génération des acquittements utilisées

Systématiquement, dès qu'un paquet a correctement été décodé, un accusé de réception est envoyé à l'émetteur pour confirmer la bonne réception du paquet.

Le principe des acquis cumulatifs aurait pu être utilisé pour réduire le nombre d'acquittements envoyés. Toutefois, ne connaissant pas le mécanisme utilisé par les `sender` et particulièrement ne sachant pas si les `sender` utilisent également le principe des acquis cumulatifs, on a préféré acquitté systématiquement chaque paquet. Ceci permettant d'assurer une certaine interopérabilité avec les `sender`.

3 Stratégie de tests

3.1 Stratégie de test utilisée

Pour s'assurer de l'efficacité de l'implémentation du `receiver`, plusieurs tests ont été menés en augmentant progressivement la difficulté.

D'une part, des tests ont été effectués sur un réseau parfait. Pour y arriver, l'émetteur et le receveur était en fait la même machine physique de sorte que les paquets prennent un chemin court et ne subissent pas ou peu de pertes.

On a ensuite augmenté progressivement la difficulté.

D'abord, la concurrence des connexions a été mise de côté et les fichiers suivants étaient envoyés :

- `dataToSend.txt` (5,9kB)
- `bigDataToSend.txt` (10,2 MB)

Ensuite, l'opération a été répétée mais en considérant cette fois la concurrence des connexions.

Pour cette première partie de tests, les résultats étaient ceux attendus.

D'autre part, des tests ont été effectués en situation réelle : l'émetteur et le receveur se trouvant sur deux réseaux locaux différents. La stratégie de test précédente a été répétée en ignorant dans un premier temps la concurrence des connexions et par la suite, en la considérant.

Pour s'assurer que la donnée reçue correspond exactement à celle envoyée, les hashs du fichier envoyé et reçu sont calculés et ensuite comparé. Si ils sont identiques, alors la donnée a parfaitement été transmise, sinon une erreur s'est produite. [1].

Dans un deuxième temps, nous avons travaillé avec le groupe 77 composé de Antoine Caytan et Zoé Schoof. Nous avons effectués des tests similaires. Dans le cas d'une connexion à la fois, nous ne constatons aucun problème. Par contre, quand nous effectuons un envoi de plusieurs fichiers assez lourd (environ 10 Mo), nous constatons une erreur d'écriture sur un fichier. Après plusieurs relectures de notre code, nous n'avons malheureusement pas trouvé de solution.

4 Évaluation quantitative

4.1 Performances de l'implémentation

Pour mesurer les performances de l'implémentation et notamment la vitesse de transfert, la stratégie suivante a été adoptée. Des fichiers de différentes tailles ont été envoyés successivement. En mesurant plusieurs fois l'opération et en prenant la moyenne, on obtient un certain temps pour chaque fichier de taille différente.

1. Pour calculer les hashs, la commande `sha512sum` a été utilisée. Exemple : `sha512sum bigDataToSend.txt`

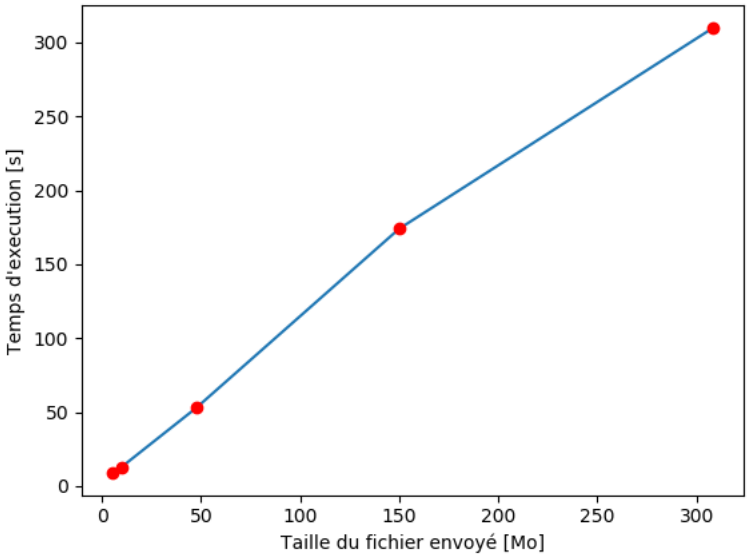


FIGURE 1 – Vitesse de transfert en fonction de la taille du fichier envoyé

4.2 Partie de l'implémentation affectant la vitesse de transfert

Pour chaque message reçu, l'adresse de l'émetteur doit être récupérée et comparée avec celles des clients actuellement connectés. Pour ce faire, la liste chaînée `ConnexionList` doit être, parfois entièrement, parcourue. Cette partie du programme influence suffisamment la vitesse de transfert.

5 Annexes

5.1 Résultat des tests d'interopérabilité et changements effectués

Grâce aux tests d'interopérabilité, une défaillance dans l'implémentation du `receiver` a été mise en évidence.

Précédemment, avant de pouvoir lire le contenu du `socket`, chacune des N connexions concurrentes autorisées devaient avoir été ouvertes.

Pour pallier cette défaillance, des changements ont été effectués. Dans la nouvelle version, un seul `socket` est utilisé pour l'écoute. La structure de données `connexion` ne comprend donc plus la variable `sfd`. Dès lors qu'un seul `socket` est ouvert pour toutes les connexions, pour pouvoir distinguer les connexions entre-elles, l'adresse du client est récupérée à chaque message reçu.

Si un client n'est pas encore connu, une structure de données `connexion` lui est attribuée et est ajoutée dans la liste chainée `ConnexionList` maintenant les différentes connexions actuellement en cours.

5.2 Pistes d'amélioration

Durant les tests, nous avons remarqué que, pour de gros fichiers reçus, la fin de fichiers que nous avons créé pouvait être différents. Cette erreur pourrait venir de l'ordre des paquets dans notre liste chainée. En effet, nous tenons compte des numéros de séquence des paquets pour mettre ceux-ci dans l'ordre d'écriture. Malheureusement, il se peut qu'avec ce choix d'implémentation l'ordre ne soit correct que pour les paquets allant de 0 à 255. Pour expliquer à l'aide d'un exemple, si nous venons de recevoir le paquet ayant le numéro de séquence 255 mais que celui-ci ne correspond pas à celui attendu (par exemple le 254), il est placé dans notre liste chaînée. Ensuite, nous recevons le paquet ayant le numéro de séquence 0. Avec notre choix d'implémentation, le paquet 0 sera stocker avant le 255 alors que l'ordre devrait être l'inverse. Une solution pour ce problème serait de stocker les paquets dans la liste chaînée grâce à un numéro de noeud qui lui serait propre. Ainsi, chaque paquet arrive dans l'ordre dans notre liste chaînée.