

# Implémentation d'un protocole de transport sans pertes

## 1 Description du projet

Ce projet est à faire par **groupe de deux**. Le but du projet est d'implémenter en langage **C** un protocole de transport utilisant des segments **UDP**. Ce protocole permettra de réaliser des transferts fiables de fichiers, utilisera la stratégie du **selective repeat**<sup>1</sup> et permettra la **truncation de payload**<sup>2</sup>. Il devra également fonctionner avec **IPv6**. Deux programmes sont à réaliser, permettant de faire un transfert de données entre deux machines distantes.

## 2 Spécifications

### 2.1 Protocole

Le format des segments du protocole est visible sur la figure 1. Ils se composent des champs dans l'ordre suivant :

**Type** Ce champ est encodé sur 2 bits. Il indique le type du paquet, trois types sont possibles :

- (i) **PTYPE\_DATA** = 1, indique un paquet contenant des données ;
- (ii) **PTYPE\_ACK** = 2, indique un paquet d'acquittement de données reçues.
- (iii) **PTYPE\_NACK** = 3, indique un paquet de non acquittement d'un packet de données tronquées (càd, avec le champ **TR** à 1).

Un paquet avec un autre type **DOIT** être ignoré ;

**TR** Ce champ est encodé sur 1 bit, et indique si le réseau a tronqué un payload initialement existant dans un packet **PTYPE\_DATA** . La réception d'un paquet avec ce champ à 1 provoque l'envoi d'un paquet **PTYPE\_NACK** . Un paquet d'un type autre que **PTYPE\_DATA** avec ce champ différent de 0 **DOIT** être ignoré ;

**Window** Ce champ est encodé sur 5 bits, et varie donc dans l'intervalle [0, 31]. Il indique la taille de la fenêtre de réception de l'émetteur de ce paquet. Cette valeur indique le nombre de places vides dans le buffer de réception de l'émetteur du segment, et peut varier au cours du temps. Si un émetteur n'a pas de buffer de réception, cette valeur **DOIT** être mise à 0. Un émetteur ne peut envoyer de nouvelles données avant d'avoir reçu un acquit pour le paquet précédant que si le champs **Window** du dernier paquet provenant du destinataire était non nul. Lors de la création d'une nouvelle connexion, l'émetteur initiant la connexion **DOIT** considérer que le destinataire avait annoncé une valeur initiale de **Window** valant 1.

**Seqnum** Ce champ est encodé sur 8 bits, et sa valeur varie dans l'intervalle [0, 255]. Sa signification dépend du type du paquet.

---

1. Le selective repeat implique uniquement que le **receiver** accepte les paquets hors-séquence et les stocke dans un buffer s'ils sont dans la fenêtre de réception. Par contre, ce protocole-ci ne permet pas au **receiver** d'indiquer au **sender** quels sont les paquets hors-séquence qu'il a reçu.

2. Tronquer les payloads peut permettre au réseau de décongestionner ses buffers tout en fournissant un mécanisme de feedback rapide. Cette idée est plus détaillée dans un article récent [3].

**PTYPE\_DATA** Il correspond au numéro de séquence de ce paquet de données. Le premier segment d'une connexion a le numéro de séquence 0. Si le numéro de séquence ne rentre pas dans la fenêtre des numéros de séquence autorisés par le destinataire, celui-ci **DOIT** ignorer le paquet ;

**PTYPE\_ACK** Il correspond au numéro de séquence du prochain numéro de séquence attendu (c-à-d (le dernier numéro de séquence + 1)  $\%2^8$ ). Il est donc possible d'envoyer un seul paquet **PTYPE\_ACK** qui sert d'acquittement pour plusieurs paquets **PTYPE\_DATA** (principe des acquis cumulatifs) ;

**PTYPE\_NACK** Il correspond au numéro de séquence du paquet tronqué qui a été reçu. Si il ne rentre pas dans la fenêtre des numéros de séquence envoyés par l'émetteur, celui-ci **DOIT** ignorer le paquet.

Lorsque l'émetteur atteint le numéro de séquence 255, il recommence à 0 ;

**Length** Ce champ est encodé sur 16 bits en network-byte order, et sa valeur varie dans l'intervalle  $[0, 512]$ . Il dénote le nombre d'octets de données dans le payload. Un paquet **PTYPE\_DATA** avec ce champ à 0 et dont le numéro de séquence correspond au dernier numéro d'acquittement envoyé par le destinataire signifie que le transfert est fini. Une éventuelle troncation du paquet ne change pas la valeur de ce champ. Si ce champ vaut plus que 512, le paquet **DOIT** être ignoré ;

**Timestamp** Ce champs est encodé sur 4 octets, et représente une valeur opaque donc sans endianness particulière. Pour chaque paquet **PTYPE\_DATA** l'émetteur d'un paquet choisit une valeur à mettre dans ce champs. Lorsque le destinataire envoie un paquet **PTYPE\_ACK** il indique dans ce champs la valeur du champs Timestamp du **dernier** paquet **PTYPE\_DATA** reçu. La signification de la valeur est laissée libre aux implémenteurs ;

**CRC1** Ce champs est encodé sur 4 octets, en network byte-order. Ce champ contient le résultat de l'application de la fonction CRC32<sup>3</sup> au header avec le champ TR mis à 0<sup>4</sup>, juste avant qu'il ne soit envoyé sur le réseau. À la réception d'un paquet, cette fonction doit être recalculée sur le header avec le champ TR mis à 0, et le paquet **DOIT** être ignoré si les deux valeurs diffèrent.

**Payload** Ce champ contient au maximum 512 octets. Il contient les données transportées par le protocole. Si le champ TR est 0, sa taille est donnée par le champs Length ; sinon, sa taille est nulle.

**CRC2** Ce champs est encodé sur 4 octets, en network byte-order. Ce champ contient le résultat de l'application de la fonction CRC32 à l'éventuel payload, juste avant qu'il ne soit envoyé sur le réseau. Ce champ n'est présent que si le paquet contient un payload et qu'il n'a pas été tronqué (champ TR à 0). À la réception d'un paquet avec ce champ, cette fonction doit être recalculée sur le payload, et le paquet **DOIT** être ignoré si les deux valeurs diffèrent.

Le modèle du réseau dans lequel ce protocole fonctionne est le suivant :

1. Un segment de données envoyé par un hôte est reçu **au plus une fois** (pertes mais pas de duplication) ;
2. Le réseau peut **corrompre** les segments de données de façon aléatoire ;

---

3. Le diviseur (polynomial) de cette fonction est  $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ , la représentation normale de ce polynôme (en network byte-order) est  $0 \times 04C11DB7$ . L'implémentation la plus courante de cette fonction se trouve dans `zlib.h`, mais de nombreuses autres existent.

4. En règle générale, un routeur commencera à tronquer les paquets lorsque ses charges réseau et/ou CPU deviendront importantes. On voudrait donc éviter que ce dernier ait à recalculer le CRC (et donc perdre du temps CPU) si le champ TR change de valeur.

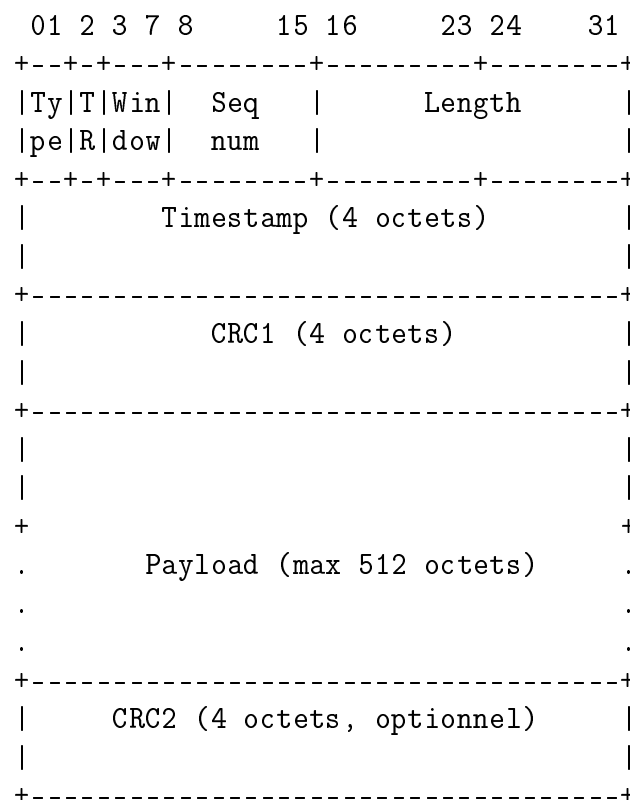


FIGURE 1 – Format des paquets du protocole

3. Le réseau peut **tronquer** le payload des paquets de façon aléatoire ;
4. Soit deux paquets,  $P_1$  et  $P_2$ , si  $P_1$  est envoyé avant  $P_2$ , il n'y a **pas de garantie** concernant l'**ordre** dans lequel ces paquets seront reçus à la destination ;
5. La **latence** du réseau pour acheminer un paquet varie dans l'intervalle **[0,2000]** (ms).

## 2.2 Programmes

L'implémentation du protocole devra permettre de réaliser un transfert de données unidirectionnel au moyen de deux programmes, **sender** et **receiver**. Ces deux programmes devront être produit au moyen de la cible par défaut d'un Makefile. Cette implémentation **DOIT** fonctionner sur les ordinateurs **Linux** de la salle Intel, bâtiment Réaumur<sup>5</sup>.

Les deux programmes nécessitent au minimum deux arguments pour se lancer :  
**sender hostname port**, avec **hostname** étant le nom de domaine ou l'adresse IPv6 du **receiver**, et **port** le numéro de port UDP sur lequel le **receiver** a été lancé.  
**receiver hostname port**, avec **hostname** le nom de domaine où l'adresse IPv6 sur laquelle le **receiver** doit accepter une connexion (:: pour écouter sur toutes les interfaces), et **port** le port UDP sur lequel le **receiver** écoute.

Enfin, ces deux programmes supporteront l'argument optionnel suivant :  
**-f X**, afin de spécifier un fichier X à envoyer par le **sender** ou le fichier dans lequel sauver les données reçues par le **receiver**. Si cet argument n'est pas spécifié, le **sender** envoie ce qu'il lit sur l'entrée standard (CTRL-D pour signaler EOF), et le **receiver** affiche ce qu'il reçoit sur sa sortie standard.

<sup>5</sup>. Les machines peuvent être accédées via SSH; consultez <https://wiki.student.info.ucl.ac.be/Mat%c3%a9riel/SalleIntel>

Les deux programmes doivent utiliser la **sortie d'erreur standard** s'ils veulent afficher des informations à l'utilisateur.

Voici quelques exemples pour lancer les programmes :

<code>sender ::1 12345 &lt; fichier.dat</code>	Redirige le contenu du fichier <code>fichier.dat</code> sur l'entrée standard, et l'envoie sur le <b>receiver</b> présent sur la même machine ( <code>::1</code> ), qui écoute sur le port 12345
<code>sender -f fichier.dat ::1 12345</code>	Idem, mais sans passer par l'entrée standard
<code>sender -f fichier.dat localhost 12345</code>	Idem, en utilisant un nom de domaine ( <code>localhost</code> est défini dans <code>/etc/hosts</code> )
<code>receiver :: 12345 &gt; fichier.dat 2&gt; log.log</code>	Lance un receiver qui écoute sur toutes les interfaces, et affiche le contenu du transfert sur la sortie standard, qui est redirigée dans <code>fichier.dat</code> , alors que les messages d'erreur et de log sont redirigés dans <code>log.log</code>

## 3 Tests

### 3.1 INGIInious

Des tests INGIInious seront fournis pour vous aider à tester deux facettes du projet :

1. La création, l'encodage et le décodage de segments ;
2. L'envoi et la réception de donnée sur le réseaux, multiplexés sur un socket.

### 3.2 Tests individuels

Les tests INGIInious ne seront pas suffisant pour tester votre implémentation. Il vous est donc demandé de tester par vous-même votre code, afin de réaliser une suite de test, et de le documenter dans votre rapport.

### 3.3 Test d'interopérabilité

Vos programmes doivent être inter-opérables avec les implémentations d'autres groupes. Vous ne pouvez donc pas créer de nouveau type de segments, ou rajouter des méta-données dans le payload. Une semaine avant la remise de la deuxième soumission, vous devrez tester votre implémentation avec 2 autres groupes (votre **sender** et leur **receiver**, votre **receiver** et leur **sender**).

## 4 Planning et livrables

### Première soumission, 23/10 à 10h

1. Rapport (max 4 pages, en PDF), décrivant l'architecture générale de votre programme, et répondant au minimum aux questions suivantes :
  - Que mettez-vous dans le champs `Timestamp`, et quelle utilisation en faites-vous ?
  - Comment réagissez-vous à la réception de paquets `PTYPE_NACK` ?
  - Comment avez-vous choisi la valeur du retransmission timeout ?
  - Quelle est la partie critique de votre implémentation, affectant la vitesse de transfert ?

— Quelle stratégie de tests avez-vous utilisée ?

2. Implémentation des deux programmes permettant un transfert de données en utilisant le protocole.
3. Suite de tests des programmes.
4. Makefile dont la **cible par défaut** produit les deux programmes dans le répertoire courant avec comme noms **sender** et **receiver**, et dont la cible **tests** lance votre suite de tests.

**Tests d'inter-opérabilité, 25/10 de 8h30 à 12h45 et/ou le 26/10 de 14h00 à 18h15, salle Intel (planning à confirmer)**

### Deuxième soumission, 3/11/2016

Même critères que pour la première condition, le rapport devant décrire en plus le résultat des tests d'interopérabilité en annexe, ainsi que les changements effectués au code si applicable.


### Format des livrables

La soumission fera en une seule archive **ZIP**, respectant le format suivant :

/	Racine de l'archive
Makefile	Le Makefile demandé
src/	Le répertoire contenant le code source des deux programmes
tests/	Le répertoire contenant la suite de tests
rapport.pdf	Le rapport
gitlog.stat	La sortie de la commande <code>git log --stat</code>

Cette archive sera nommée `projet1_nom1_nom2.zip`, où `nom1/2` sont les noms de famille des membres du groupe, et sera à mettre sur Moodle.

Il est demandé de réaliser le projet sous un gestionnaire de version (typiquement `git`).

 **Important** : l'évaluation tiendra compte de vos deux soumissions, et pénalisera les projets qui ont deux soumissions trop différentes. La première soumission n'est donc **PAS** facultative.

## 5 Evaluation

La note du projet sera composée des trois parties suivantes :

1. Implémentation : Vos programmes fonctionnent-ils correctement ? Qualité de la suite de tests ? Que se passe-t-il quand le réseau est non-idéal ? Sont-ils inter-opérables ? ...  
Le respect des spécifications (arguments, fonctionnement du protocole, formats des segments, structure de l'archive, ...) est impératif à la réussite de cette partie !
2. Rapport ;
3. Code review individuelle du codes de deux autres groupes. Effectuée durant la semaine suivant la remise du projet. Vous serez noté sur la pertinence de vos review.

## 6 Ressources utiles

Les manpages des fonctions suivantes sont un bon point de départ pour implémenter le protocole, ainsi que pour trouver d'autres fonctions utiles : `socket`, `bind`, `getaddrinfo`, `connect`, `send`, `sendto`, `sendmsg`, `recv`, `recvfrom`, `recvmsg`, `select`, `poll`, `getsockopt`, `shutdown`, `read`, `write`, `fcntl`, `getnameinfo`, `htonl`, `ntohl`, `getopt`

[2] est un tutoriel disponible en ligne, présentant la plupart des appels systèmes utilisées lorsque l'on programme en C des applications interagissant avec le réseau.

[8] et [4] sont deux livres de références sur la programmation système dans un environnement UNIX, disponibles en bibliothèque INGI.

[1] est le livre de référence sur les sockets TCP/IP en C, disponible en bibliothèque INGI.

[7] et [6] sont deux mini-tutoriels pour la création d'un serveur et d'un client utilisant des sockets UDP.

[5] et `man select_tut` donnent une introduction à l'appel système `select`, utile pour lire et écrire des données de façon non-bloquante.

## Références

- [1] Michael J. Donahoo and Kenneth L. Calvert. *Tcp / ip sockets in c, a practical guide for programmers*, 2001.
- [2] Brian "Beej Jorgensen" Hall. *Beej's guide to network programming*, 2015. URL : <https://beej.us/guide/bgnet/output/html/singlepage/bgnet.html>.
- [3] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42. ACM, 2017, available at <http://conferences2.sigcomm.org/sigcomm/2017/program.html>.
- [4] Michael Kerrisk. *The linux programming interface : a linux and unix system programming handbook*, 2010.
- [5] Spencer Low. The world of `select()`. URL : <http://www.lowtek.com/sockets/select.html>.
- [6] Graham Shaw. Listen for and receive udp datagrams in c. URL : [http://www.microhowto.info/howto/listen\\_for\\_and\\_receive\\_udp\\_datagrams\\_in\\_c.html](http://www.microhowto.info/howto/listen_for_and_receive_udp_datagrams_in_c.html).
- [7] Graham Shaw. Send a udp datagram in c. URL : [http://www.microhowto.info/howto/send\\_a\\_udp\\_datagram\\_in\\_c.html](http://www.microhowto.info/howto/send_a_udp_datagram_in_c.html).
- [8] W. Richard Stevens and Stephen A. Rago. *Advanced programming in the unix environment*, 2005.