Protocole de transport fiable
Alternating-Bit et Go-Back-N
Sys 1 : Programmation réseau (et système)

Après s'être intéressé à la couche applicative, vous allez maintenant mettre les mains dans la couche transport. L'objectif de ce projet est d'écrire deux versions d'un protocole de transport fiable.

Objectifs pédagogiques:

- Mettre en pratique la manipulation d'un protocole encapsulant un niveau supérieur;
- Comprendre et mettre en pratique un protocole de transport fiable (au moins Alternating Bit et si possible un mécanisme fenêtré).

★ Environnement de travail

Le code que vous devez écrire simulera de manière proche le fonctionnement d'un protocole dans une situation réelle. Autrement dit, il devra s'insérer entre la couche applicative et la couche réseaux. Pour ce faire, un environnement qui simule ces deux couches vous est fourni. De même, les timings des requêtes, de latence et d'interruptions seront également simulées aléatoirement. Un schéma de l'environnement vous est donné en Figure 1.

■ Ce que vous devez écrire

Les procédures que vous devez écrire permettront à un émetteur (A) de discuter avec un récepteur (B). Vous n'avez besoin d'implémenter qu'une version unidirectionnelle (depuis A vers B). Attention, l'acteur B enverra bien entendu des paquets pour "acknowledge" les échanges. Comme indiqué dans sur la figure, vos procédures seront appelées et devront appeler des fonctions qui simulent d'une part, la couche applicative et d'autre part, un medium d'échange de donnée pouvant perdre, délayer voir corrompre les paquets échangés.

Les données échangées entre la couche applicative et votre protocole seront des *messages*, qui correspondent à la structure C suivante :

```
1  typedef struct {
2    char data[20];
3  } msg_t;
```

Votre émetteur va donc recevoir des données par bloc de 20 octets depuis la couche 5. Votre récepteur doit ainsi délivrer des données *correctes* par bloc de 20 octets à la couche 5 de son côté.

Les données échangées entre votre protocole et la couche de réseaux seront des *paquets*, qui correspondent à la structure C suivante :

```
1  typdedef struct {
2    int seqnum;
3    int acknum;
4    int checksum;
5    char payload[20];
6  } pkt_t;
```

Votre protocole devra remplir le champ `payload` à l'aide des données reçu depuis la couche 5. Les autres champs devront être utilisés pour assurer la fiabilité de la transmission comme vu en cours.

Les différentes fonctions que vous devez écrire sont détaillés après. Dans la réalité, ces fonctions sont en parties implémentées directement dans votre système d'exploitation et utilisée par certains appels systèmes.

- `A_output(msg_t msg)` où `msg` est un message contenant les données à envoyer à l'acteur B. Cette fonction sera appelée à chaque fois que l'application côté A souhaitera envoyer un message à l'application côté B. C'est votre rôle d'assurer que le message soit transmis correctement.
- `A_input(pkt_t pkt)` est appelée à chaque fois qu'un paquet `pkt` potentiellement corrompu, qui a été émis par l'acteur B (via `tolayer3(B, pkt)`), arrive du côté de l'acteur A.

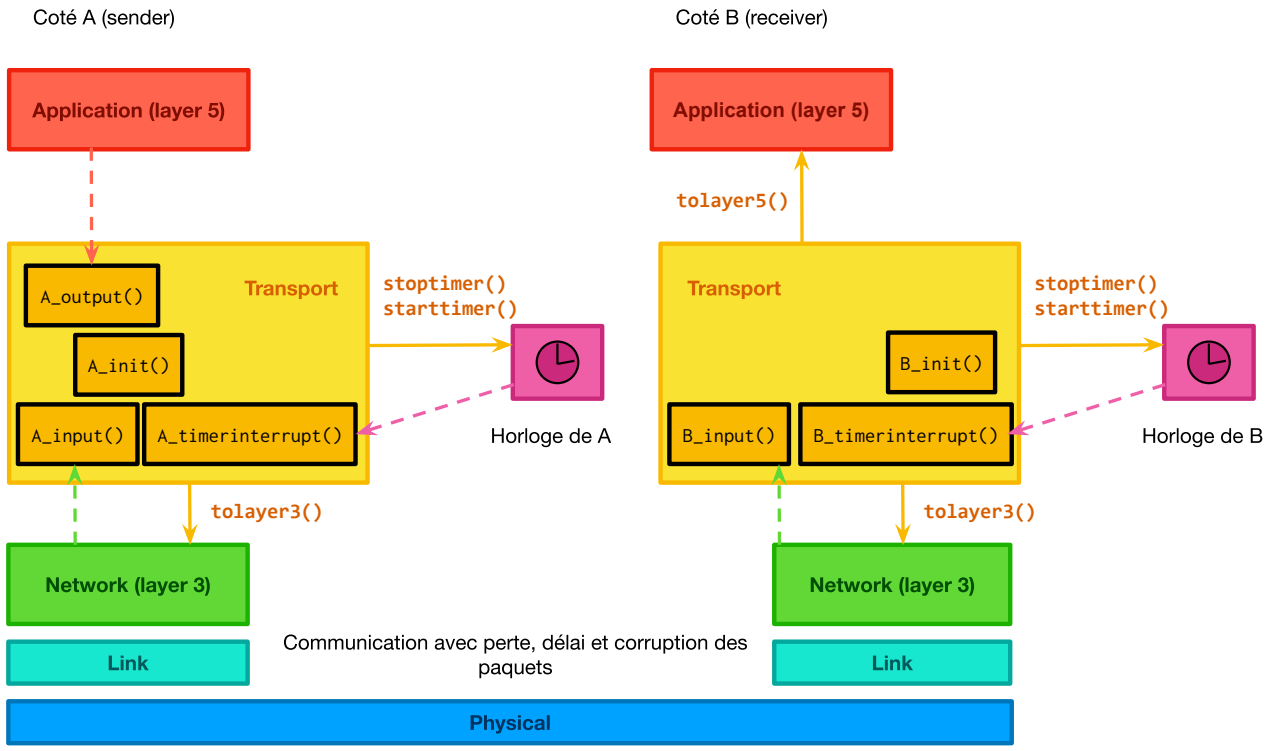

Figure 1: Structure de l'environnement : en noir les fonctions que vous devez implémenter. Les flèches pleines représentent l'interface mise à votre disposition tandis que les flèches en pointillés correspondent aux appels de vos fonctions par le simulateur.

- `B_input(pkt_t pkt)` est appelée lorsque qu'un paquet `pkt` potentiellement corrompu, qui a été émis par l'acteur A (via `tolayer3(A, pkt)`), arrive du côté de l'acteur B.
- `A_timerinterrupt()` (resp. `B_timerinterrupt()`) est appelée lorsque l'horloge de A (resp. de B) expire (simulant ainsi une interruption "timeout"). Vous aurez certainement besoin d'interagir avec l'horloge via les fonctions `starttimer()` et `stoptimer()`pour gérer la retransmission de paquets.
- `A_init()` et `B_init()` sont appelées une fois au démarrage de la simulation et peut donc être utilisée pour initialiser de potentielles données.

L'ensemble des déclarations de ces structures ainsi que de ces fonctions sont disponibles dans le fichier `src/protocol_routines.h`.

■ L'interface mis à disposition

Pour écrire les fonctions spécifiées précédemment, vous pourrez utiliser des fonctions mises à votre disposition pour interagir avec l'environnement.

- `starttimer(int AorB, double increment)` permet de lancer l'horloge de l'acteur `AorB`, qui serra interrompu dans `increment` unités de temps. `starttimer(A, 5.0)` indique par exemple qu'à moins que l'acteur A n'interrompe son horloge avant, la fonction `A_timerinterrupt()` sera appelée dans 5 unités de temps. Pour vous donner une idée des valeurs adéquates, un paquet seul sur le réseau met en moyenne 5 unités de temps pour traverser la couche physique. Les horloges de A et de B ne devraient être appelées que par leur propriétaire respectif.
- `stoptimer(int AorB)` permet d'interrompre l'horloge de l'acteur `AorB`.
- `tolayer3(int AorB, pkt_t* packet)` permet d'envoyer un paquet `packet` à l'autre entité via le réseau.
- `tolayer5(int AorB, msg_t* message)` permet d'envoyer un message `message` à l'application de l'acteur `AorB`.

Les déclarations de ces fonctions sont disponibles dans le fichier `simulator/softare_interfaces.h`. Notez que vous pouvez, si vous le désirez, inspecter l'implémentation réalisée dans le fichier C correspondant, mais **il n'est pas nécessaire de le faire** pour mener le projet à bien.

■ La simulation du réseau

Le reste du code fourni permet de simuler les couches inférieures. Encore une fois, vous pouvez inspecter le code si cela vous intéresse. Cependant, vous pouvez tout aussi bien réaliser l'entièreté du projet en considérant la simulation comme une boite noire. Les seules choses à savoir sur le réseau de la simulation sont les suivantes :

- Les paquets en transits peuvent être perdus et corrompus, mais ne seront jamais réordonnés (FIFO Channel). Autrement dit, si 2 paquets traversent la couche réseaux, alors l'ordre d'émission sera le même que l'ordre de réception.

De plus, certains paramètres vous seront demandés au début de la simulation :

- **Le nombre de messages à simuler.** L'environnement s'arrêtera dès que ce nombre de messages aura été émis depuis la couche 5. Vous n'avez **pas** à vous inquiéter du bon délivrement des paquets au-delà de l'émission du dernier message. En particulier, si cette valeur est réglée à 1, le programme s'arrêtera dès l'émission du premier message (autrement dit immédiatement). Si l'environnement n'a pas le temps de tester suffisamment vos fonctions, n'hésitez pas à augmenter ce paramètre.
- **Le taux de perte de paquet.** La probabilité qu'un paquet soit perdu sur le réseau.
- **Le taux de corruption.** La probabilité qu'un paquet non perdu soit corrompu. Notez qu'une corruption signifie qu'un des quatre champs de la structure pkt_t a été altéré.
- **Délai moyen entre deux messages de l'application.** N'importe quelle valeur strictement positive est valide.
- **Le niveau de tracing.** Indique le niveau de débogage attendu. Une valeur de 1 ou 2 affichera des informations additionnelles sur les paquets échangées par le réseau. Une valeur de 0 désactive les sorties tandis qu'une valeur supérieure à 2 affiche des informations de débogage du simulateur lui-même (si vous pensez en avoir besoin, pensez-y au moins une deuxième fois avant).

Pour les parties suivantes, vous devrez écrire les fonctions attendues dans le fichier de protocole correspondant qui se situe dans le dossier `src`. Un `CMakeFile` vous est fourni à la racine du projet. Il produit deux exécutables `Mini_TCP_Alternating` et `Mini_TCP_goBackN` correspondant respectivement aux parties 1 et 2 du sujet.

★ Première partie : Le Alternating-Bit-Protocol

Dans cette partie, vous devez implémenter le protocole en bit alternant vu en cours en utilisant à la fois des ACK et NACK.

Pour tester vos fonctions n'hésitez pas à choisir un délai entre deux messages d'application suffisamment grand afin d'être sûr qu'il n'y a pas de messages non validés en vol lorsque la fonction `A_output()` est appelée. Une valeur de 1000 devrait être suffisante. En particulier, nous vous conseillons de vérifier lors de l'appel à `A_output()` si un message est encore en cours de transmission par votre protocole. Si tel est le cas, vous pourrez pour l'instant ignorer le message de l'application.

▷ **Question 1:** Implémentez les fonctions du fichier `src/protocol_alternating_bit.c` pour mettre en œuvre le protocole attendu.

Enrichissez votre protocole pour qu'il affiche les différents évènements ayant lieu (réception d'un message ou d'un paquet, détection d'une corruption, etc.) ainsi que les réponses effectuées (émission d'un message ou d'un paquet, renvoie d'un ACK, etc.).

▷ **Question 2:** Éxecutez votre protocole de sorte que 10 messages soient reçus par B, avec une probabilité de perte de 0.1, une probabilité de corruption de 0.3 et un affichage de débogage de niveau 2. Commentez la sortie obtenue afin d'expliquer pour au moins une perte et au moins une corruption comment votre protocole a réussi à rester fiable.

★ (Vraiment Optionnel) Deuxième partie : Le Go-Back-N-Protocol

Dans cette partie, vous devez implémenter le protocole Go-Back-N avec une fenêtre de taille 8. Cette partie est une extension de la précédente. Autrement dit, assurez-vous d'avoir bien compris la partie précédente et d'avoir un code robuste avant de vous attaquer à cette version. Quelques considérations à prendre en compte par rapport à la version précédente :

- `A_output(msg_t msg)` peut désormais être appelé alors qu'un certain nombre de paquets est encore en train d'être échangé via le réseau. Ainsi, il vous faudra dorénavant pouvoir stocker un certain nombre de messages en attente. Cependant, vous pourrez vous contenter d'avoir un nombre maximum fixe de message que vous autoriserez à avoir en attente coté émetteur (100 devrait être suffisant pour nos simulations). Dans le cas où vous viendriez à dépasser les 100 buffers dont vous disposez, vous pourrez simplement arrêter la simulation avec un `exit()`. Dans une vraie implémentation, notez qu'il faudrait bien entendu trouver une solution plus élégante.
- `A_timerinterrupt()`. Notez que vous ne disposez que d'une unique horloge alors que plusieurs paquets peuvent être en vol en parallèle. Il vous faudra donc réfléchir à la question d'utiliser au mieux cet unique horloge.

▷ **Question 3:** Implémentez les fonctions du fichier `src/protocol_go_back_n.c` pour mettre en œuvre le protocole attendu.

Enrichissez votre protocole pour qu'il affiche les différents évènements ayant lieu (réception d'un message ou d'un paquet, détection d'une corruption, etc.) ainsi que les réponses effectuées (émission d'un message ou d'un paquet, renvoie d'un ACK, etc.).

▷ **Question 4:** Éxecutez votre protocole de sorte que 20 messages soient reçus par B, avec une probabilité de perte de 0.2, une probabilité de corruption de 0.2, un temps moyen entre deux messages applicatif de 10 et un affichage de débogage de niveau 2. Commentez la sortie obtenue afin d'expliquer pour au moins une perte et au moins une corruption comment votre protocole a réussi à rester fiable.

★ Remarques, conseils et attendu du rendu

- **Somme de contrôle.** Vous êtes libre d'utiliser la méthode que vous souhaitez. N'oubliez pas que le numéro de séquence ainsi que le champ d'ACK peuvent également être corrompus. Vous pouvez utiliser une somme de contrôle semblable à TCP qui somme les champs de séquence, d'ACK ainsi que chacun des caractères du packet, traités comme des entiers.
- Notez que n'importe quelle donnée partagée entre deux appels de vos fonctions doit être manipulée à l'aide de variable globale (ou statique). Par exemple, vos fonctions risquent de devoir garder une copie d'un paquet pour une potentielle retransmission. Une telle donnée pourrait très bien être une variable globale de votre code. Cependant, gardez bien en tête que de telles variables ne doivent d'être accessibles que par un unique acteur à la fois ! Le seul moyen pour les acteurs A et B de s'échanger des informations réside dans le réseau.
- Au besoin, vous pourrez afficher le contenu de la variable flottante `time` qui est défini en tant que globale et correspond au temps actuel au sein de la simulation.
- **Commencez simplement.** Réglez les probabilités de perte et de corruption à 0 et assurez-vous que vos fonctions marchent dans ce cas-là. N'hésitez pas à `git commit` les versions fonctionnelles de votre code au fur à mesure pour pouvoir annuler plus facilement un changement un peu trop ambitieux. Rajouter petit à petit la possibilité de perte, puis de corruption et finalement des deux en même temps.
- **Débogage.** Le simulateur est là pour vous aider en affichant des informations sur les paquets en vols sur le réseau. N'hésitez pas à rajouter tout affichage qui pourrait vous être utile dans votre code.

Pour le rendu de ce projet, nous attendons a minimia vos fichiers `src/protocol_alternating_bit.c` (et `src/protocol_go_back_n.c`, le cas échéant) nettoyés ainsi que vos réponses aux questions 2 (et 4 le cas échéant) sous format texte (markdown, pdf, txt ou tout autre format vous permettant d'avoir la sortie de votre programme accompagnée de vos explications). Un lien git vers votre projet est une possibilité[1], mais n'est pas la seule.

[1] avec l'envoie avant la deadline de l'identifiant (hash) du commit rendu