TP réseaux : développement et test de protocoles de transfert fiable de données

1 Présentation générale du projet

L'objectif de ce projet est de développer (en langage C) des protocoles de transfert de données mettant en œuvre différents mécanismes de gestion de la fiabilité (contrôle d'erreurs, contrôle de flux, retransmissions). Ces protocoles de transfert de données seront vus dans le contexte d'une couche transport (même schéma que celui vu en Cours/TD). Dans la suite du sujet, nous considérerons uniquement 3 couches : application, transport et réseau. La figure ci-dessous illustre l'articulation de ces trois couches :

— La **couche application** intègre un protocole de transfert de fichiers entre un émetteur et un récepteur. Cette couche a déjà été développée, elle vous est fournie sous la forme d'un module dont l'appel se fait par des fonctions présentées dans la section 2.

— La **couche transport** constitue le cœur du projet. Vous devrez développer dans cette couche plusieurs protocoles de transfert de données (unidirectionnels pour les données) supportant différents types de services détaillés dans la section 4.

— La **couche réseau** a également déjà été implémentée et vous est fournie sous la forme d'un module. Elle peut fonctionner soit en mode local (émetteur et récepteur sur la même machine), soit en mode réparti (émetteur et récepteur sur des machines distinctes). Elle est détaillée dans la section 3.

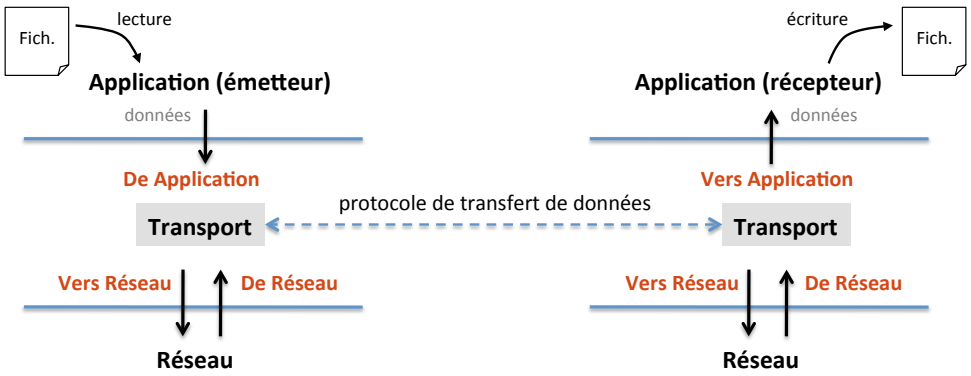

FIGURE 1 – Positionnement des couches *Application*, *Transport*, *Réseau* et de leurs interactions

2 Interactions avec la couche application

Un protocole de transfert de fichiers entre un émetteur et un récepteur est implémenté dans la couche application. L'émetteur lit un fichier de données bloc par bloc, chacun d'eux est remis à la couche transport pour transfert. Le récepteur récupère les blocs de données de sa couche transport et les écrit dans le fichier de destination. Le code de la couche application se trouve dans le fichier `couche_appli_non_connectee.c` dans laquelle l'application s'appuie sur les services de la couche transport en mode non connecté.

L'interface de programmation entre la couche transport et la couche application est spécifiée dans le fichier `application.h` et contient deux fonctions : `de_application` pour récupérer des données ou indications issues de la couche application et `vers_application` pour les lui remettre. Nous vous listons ci-dessous ces deux fonctions :

```
/*************************************
 * Interface avec la couche application *
 *************************************/

/*
 * Lecture de données émanant de la couche application.
 * Paramètres (en sortie):
 *  - donnees : message devant être envoyé (issu de la couche application)
 *  - taille_msg : nombre d'octets de données à émettre (0 si pas de données)
 */
void de_application(unsigned char *donnees, int *taille_msg);

/*
 * Remise de données à la couche application.
 * Paramètres (en entrée):
 *  - donnees : données à remonter à l'application
 *  - taille_msg : taille des données
 * Renvoie :
 *    -> 1 si le récepteur n'a plus rien à écrire (fichier terminé)
 *    -> 0 sinon
 */
int vers_application(unsigned char *donnees, int taille_msg);
```

3 Interactions avec la couche réseau

Vous disposez d'une bibliothèque (`services_reseau.h`) qui vous offre un service de transfert de paquets non fiable entre une source et une destination. Pour utiliser cette bibliothèque, vous devez commencer par appeler la fonction d'initialisation `init_reseau()`. Puis, vous utilisez les primitives `vers_reseau()` et `de_reseau()` pour envoyer et recevoir des paquets. Enfin, vous disposez de fonctions utilitaires pour la gestion de temporisateurs (*timers*) et l'attente d'un événement (*timeout* ou paquet arrivé).

Initialisation couche réseau

```
/*********************************************************
 * Initialisation de le couche réseau. Le paramètre rôle *
 * doit être égal à 1 pour la réception, 0 pour l'émission. *
 *********************************************************/
void init_reseau(int role);
```

Primitives de service pour émission et réception sur le canal

```
/*************************************************
 * Remet un paquet à la couche réseau pour envoi *
 * sur le support de communication.              *
 *************************************************/
void vers_reseau(paquet_t *paquet);

/*********************************************************
 * Prélève un paquet de la couche réseau (N.B. : fonction *
 * bloquante tant qu'un paquet n'est pas reçu).          *
 *********************************************************/
void de_reseau(paquet_t *paquet);
```

Fonctions utilitaires pour la gestion des temporisateurs (*timers*)

```
/*****************************************************************
 * Démarre le temporisateur numéro n (0 <= n <= 31), qui s'arrêtera  *
 * après ms millisecondes (ms doit être un multiple de 100)          *
 *****************************************************************/
void depart_temporisateur(int n, int ms);

/*********************************
 * Arrête le temporisateur numero n *
 *********************************/
void arreter_temporisateur(int n);

/*************************************************************
 * Attend un évènement : paquet reçu ou timeout.             *
 * (N.B. : fonction bloquante)                               *
 * Retour : -1 si un paquet reçu est disponible, sinon       *
 *          le numéro de temporisateur qui a expiré          *
 *************************************************************/
int attendre();
```

4 Travail à réaliser

Le travail demandé consiste à implémenter les protocoles (0 à 4) décrits ci-dessous. Pour chaque cas, vous développerez un programme principal pour l'émetteur et un programme principal pour le récepteur, qui devront respecter la convention de nommage décrite dans les annexes.

Pour implémenter vos protocoles, vous vous appuierez sur le type de données `paquet_t` spécifié ci-dessous. Pour simplifier, le même format de paquet sera utilisé pour tous les protocoles. Certains protocoles n'utiliseront donc pas tous les champs de la structure. Le type `paquet_t` ainsi que d'autres constantes sont définis dans le fichier `couche_transport.h`.

```
/************************
 * Structure d'un paquet *
 ************************/
typedef struct paquet_s {
    uint8_t type;         /* type de paquet, cf. constantes dans .h */
    uint8_t num_seq;      /* numéro de séquence */
    uint8_t lg_info;      /* longueur du champ info */
    uint8_t somme_ctrl;   /* somme de contrôle */
    unsigned char info[MAX_INFO];  /* données utiles du paquet */
} paquet_t;
```

4.1 Protocole v0 : transfert de données sans garantie

Avec ce protocole, le service offert n'apporte aucune garantie sur la délivrance des données (des erreurs ou pertes peuvent se produire). Aucun mécanisme de contrôle de flux, ni de contrôle et reprise sur erreurs n'est implémenté.

4.2 Protocole v1 : transfert de données avec détection d'erreurs et contrôle de flux « Stop-and-Wait »

Cette version de protocole rajoute deux mécanismes :
— un contrôle de flux en mode « Stop-and-Wait ».

— une détection d'erreurs basée sur une somme de contrôle. Cette somme sera calculée en appliquant l'opérateur « ou exclusif » (XOR) sur trois octets de l'en-tête (`type`, `num_seq`, `lg_info`) ainsi que sur tous les octets de données du paquet. Si aucune erreur n'est détectée par le récepteur alors un acquittement positif sera renvoyé (type ACK), sinon un acquittement négatif (type NACK).

Notez qu'on ne considérera dans cette version que des erreurs bits sur des paquets de type DATA (i.e. émetteur vers récepteur), pas sur les paquets d'acquittement.

4.3 Protocole v2 : transfert de données avec contrôle de flux et reprise sur erreurs « Stop-and-Wait »

La gestion des pertes sera rajoutée dans ce protocole (en plus des erreurs bits), avec une reprise sur erreurs « Stop-and-Wait ». Contrairement à la version précédente, la retransmission se fera suite à l'expiration d'un temporisateur (pas d'acquittement négatif).

4.4 Protocole v3 : transfert de données avec fenêtre d'anticipation et reprise sur erreurs « Go-Back-N »

Ce protocole utilise une fenêtre d'anticipation avec une stratégie de retransmissions de type « Go-Back-N ». La capacité de numérotation sera de 8 ou 16 et la fenêtre d'émission aura une taille strictement inférieure à cette capacité (la taille de la fenêtre pourra être définie à l'appel du programme). Rappels des points importants sur le « Go-Back-N » :
Pour l'émetteur :

— un seul temporisateur est suffisant pour toute sa fenêtre ;
— à la réception d'un ACK sans erreur, si le numéro d'ACK est compris dans la fenêtre, l'émetteur décale sa fenêtre de manière cumulative ;
— lors d'un *timeout*, l'émetteur retransmet toute sa fenêtre (jusqu'à son pointeur courant).

Pour le récepteur :

— il n'accepte que les paquets en séquence (fenêtre de réception=1) ;
— chaque paquet reçu sans erreur génère un ACK. Si le paquet reçu est en séquence alors on acquitte ce paquet, sinon on acquitte le dernier paquet reçu correctement.

Notez que pour ce protocole, nous vous fournissons déjà la fonction `dans_fenetre()` pour vérifier si un numéro de séquence est inclus dans la fenêtre d'émission (cf. `couche_transport.h`)

4.5 Protocole v4 : transfert de données avec fenêtre d'anticipation et reprise sur erreurs « Selective Repeat »

Ce protocole utilise une fenêtre d'anticipation avec une stratégie de retransmissions de type « Selective Repeat ». La capacité de numérotation sera de 8 ou 16 et la fenêtre d'émission aura une taille maximale de la moitié de cette capacité (la taille de la fenêtre pourra être définie à l'appel du programme). Rappels des points importants sur le « Selective Repeat » :
Pour l'émetteur :

— un temporisateur pour chaque paquet envoyé devra être géré ;
— à la réception d'un ACK sans erreur inclus dans la fenêtre, on ne décale la fenêtre que si cela est possible ;
— lors d'un *timeout*, l'émetteur ne retransmet que le paquet pour lequel le délai a expiré.

Pour le récepteur :

— il peut accepter des paquets dans la fenêtre mais hors séquence (la taille de la fenêtre de réception est égale à la taille de la fenêtre d'émission), dans ce cas ces paquets sont « bufferisés » ;
— chaque paquet reçu sans erreur génère un ACK. On acquitte le paquet qu'il soit en séquence ou hors séquence.

5 Evaluation

Sur les 6 séances de TP, vous disposez de 5 séances de travail et la 6$^{\text{ème}}$ séance sera dédiée à la validation (notez que la validation pourra aussi se faire au fur et à mesure de votre avancement).

Lors de la validation, vous ferez une démonstration à l'enseignant du transfert de fichier en utilisant les différents protocoles que vous aurez implémentés. **Une archive de votre code source devra être déposée sur moodle au plus tard lors de la dernière séance de TP**.

À titre indicatif, le barème sera le suivant :

— Protocoles : 16 pts (v1=v2=v3=v4=4pts)
— Suivi : 2 pts (assiduité, avancement, etc.)
— Code : 2 pts (lisibilité, modularité, commentaires, etc.)

Intégrité académique

Le travail doit être réalisé individuellement. Vous pouvez discuter du sujet avec vos camarades mais il est strictement interdit de copier-coller du code. L'équipe pédagogique utilise des outils de mesures de similarité de codes sources. **Tout plagiat sera sévèrement sanctionné**.

6 Annexes

6.1 Génération des exécutables `emetteur` et `recepteur`

Vos programmes pour l'émetteur et le récepteur devront respecter la convention de nommage suivante (exemple avec le protocole de transfert de données v0) :

— `proto_tdd_v0_emetteur.c` : *main()* de l'émetteur pour le protocole v0.
— `proto_tdd_v0_recepteur.c` : *main()* du récepteur pour le protocole v0.

Nous vous fournissons un *Makefile* pour effectuer la compilation et l'édition de liens. Exemple d'un cycle de travail classique :

1. `make clean` : suppression des exécutables et fichiers objets.
2. `make tdd0` pour générer les exécutables du protocole v0 dans dossier `bin/`
3. (correction du code source si erreurs éventuelles jusqu'au succès du *make*)
4. Edition des paramètres de configuration dans `config.txt` (cf. ci-dessous).
5. Test : `bin/recepteur` pour exécuter le processus de réception et `bin/emetteur` pour exécuter le processus d'émission (dans un terminal différent !).
6. Comparez les fichiers émis et reçus...

6.2 Fichier de configuration `config.txt`

Le fichier `config.txt` contient tous les paramètres de configuration de l'application. **Il est important de faire varier ces paramètres afin de tester vos protocoles**.

Paramètres qui configurent le nom des fichiers manipulés par la couche application :
— `FICHIER_IN` indique le fichier lu par l'émetteur.
— `FICHIER_OUT` indique le nom du fichier dans lequel seront écrites les données reçues.

Paramètres qui contrôlent le comportement de la couche réseau :
— `PROBA_PERTE_E` et `PROBA_PERTE_R` pour indiquer le taux de pertes respectivement côté émetteur et récepteur. Cette valeur est comprise entre 0 et 1 (notez qu'une probabilité au dessus de 0.5 est déconseillée).
— `PROBA_ERREUR_E` et `PROBA_ERREUR_R` pour indiquer le taux d'erreurs respectivement côté émetteur et récepteur. Cette valeur est comprise entre 0 et 1 (notez qu'une probabilité au dessus de 0.5 est déconseillée).

6.3 Test du transfert de données en mode « réparti » (émetteur et récepteur sur machines différentes)

On suppose l'émetteur sur la machine A, le récepteur sur la machine B. Exécutez les trois opération suivantes :

1. Récupérez les adresses IP de A et B (sur Linux, commande `ifconfig` ou `ip addr show`). Dans la suite, **remplacez @IP_A et @IP_B par vos propres adresses !**
2. Sur B, remplacez la ligne `init_reseau(RECEPTION)` par :
   `init_reseau_mode_reparti(RECEPTION, 5000, "@IP_A", 4000)`
3. Sur A, remplacez la ligne `init_reseau(EMISSION)` par :
   `init_reseau_mode_reparti(EMISSION, 4000, "@IP_B", 5000)`