Téléinformatique & Protocoles

Projet 2 : Réalisation d'un protocole de la famille HDLC

1 But du projet

Le but de ce projet est de vous faire concevoir et programmer un protocole de communication au niveau de la couche liaison de données. Tout ce qui concerne les couches supérieures et inférieures vous est fourni.

Délai : le rapport est à rendre pour le 15 janvier. La validation aura lieu à une date précisée ultérieurement.

2 La situation de référence et le travail demandé

La couche liaison n'est définie que par les services qu'elle doit rendre à une application cliente et par les services que lui offre la couche physique. Il est donc important de décrire non seulement la couche liaison mais aussi son contexte. Cette couche liaison peut être vue comme un serveur qui réagit en fonction des événements qui surviennent (réception de messages, réception d'ordres, *timeout*, etc.).

2.1 Le support de transmission

On dispose d'un support de transmission (medium) qui relie les deux processus qui communiquent. Le support est capable de transmettre dans les deux sens (full-duplex). Ce support n'est pas fiable et peut introduire des erreurs, des pertes, voire des trames parasites. Du point de vue de la couche liaison, on n'accède à ce support qu'au travers de la couche physique.

2.2 La couche physique

Dans les deux stations en communication, il existe une couche physique qui peut lire les signaux véhiculés sur le support de transmission et peut rendre à la couche liaison une suite de bits constituant une trame.

En pratique, la couche physique assure les services nécessaires aux moyens de deux fonctions qui vous sont fournies (section 8.2.1).

2.3 Les applications

Les applications se limiteront, dans le cadre de ce projet, à l'envoi et à la réception de fichiers. Pour ce faire, vous taperez des instructions d'envoi de fichiers dans des fenêtres-terminaux (sections 5 et 8.1). Une instruction d'envoi de fichiers générera un *événement* au niveau de la couche liaison.

La couche application aura pour travail, selon les cas, soit de lire un fichier et de l'émettre sur la liaison, soit de recevoir des paquets de la couche liaison et de reconstituer le fichier correspondant, soit de faire les deux à la fois. Pour la réception, nous supposerons que l'application est toujours prête à enregistrer les données que lui délivre la couche liaison.

2.4 La couche liaison

Dans un monde idéal, c'est-à-dire si le support de transmission était parfait[1], le travail de la couche liaison serait :

– pour l'émission, de mettre les paquets fournis par l'application dans des trames et d'envoyer ces trames sur la couche physique ;
– pour la réception, d'extraire les paquets contenues dans les trames reçues de la couche physique et de délivrer ces paquets à l'application.

Mais le support de transmission introduit des erreurs, des pertes, des retards, etc. La couche liaison doit donc fiabiliser le support de transmission pour l'application. Elle doit en effet assurer qu'elle délivre tous les paquets à l'application, sans erreurs de transmission et dans leur ordre d'émission. Pour ce faire, des codes de détection d'erreurs sont introduits dans les trames (les fonctions de codage, de décodage, et de détection d'erreurs vous sont fournies). S'il s'avère impossible de transférer des données sans erreurs, la couche liaison doit avertir l'application que le support de transmission est inutilisable.

Travail demandé

Le protocole réalisé par la couche liaison de données sera de la famille HDLC. Vous pouvez et vous ne devez donc utiliser que des trames définies par la norme, c'est-à-dire les trames de type : SABM, UA, DISC, FRMR,DM, INFO, RNR, auxquelles s'ajoutent : RR et REJ pour le protocole *GoBack-N* et SREJ pour le *rejet sélectif*. Ceci étant dit, l'interprétation de la signification exacte de chacun des types de trames, ainsi que les différents champs constitutifs de ces trames, sont laissés à votre libre arbitre. Autrement dit, étant données les contraintes, il vous appartient de mettre au point un protocole effectif. La contrainte essentielle étant que la couche liaison transmette les données à l'application sans erreur et dans l'ordre dans lequel elles ont été émises.

Le but du projet est de réaliser un protocole de type *rejet sélectif* prévoyant, entre autres, une phase de connexion, et une phase de déconnexion. Si l'application a quelque chose à émettre, la couche liaison commence par demander une connexion. Si l'application n'a rien à émettre, elle attend que la station distante lui demande une connexion ou que l'application lui demande d'envoyer un fichier.

3 Évaluation

L'évaluation de ce travail sera faite sur la base d'une validation, d'un rapport écrit et des programmes eux-mêmes.

3.1 Le programme

Comme pour le premier projet, nous vous demandons de nous fournir par courrier électronique, le code du programme réalisant le protocole HDLC.

3.2 Validation

Ce projet fera l'objet d'une validation devant un examinateur. Vous devrez faire fonctionner votre programme dans différentes situations : émission seule, réception seule, émission et réception de fichiers de mêmes tailles ou de tailles différentes. Pour la validation et afin de tester la robustesse de votre protocole, l'examinateur fournira un nouveau programme medium qui générera des erreurs différentes (rafales de 16 bits, etc.).

3.3 Rapport

Nous vous demandons également de nous rendre un rapport écrit décrivant votre travail. Il serait inutile d'inclure le code dans le rapport. Indépendamment de la qualité de vos programmes, le rapport devrait permettre de savoir comment vous avez réalisé le travail, sans qu'il soit nécessaire de se référer directement à votre code. En particulier ce rapport doit contenir le protocole décrit, soit avec des automates, soit avec un pseudo-code.

Quelques questions que vous devez vous poser et auxquelles vous devrez répondre dans le rapport :

[1] Cette condition est nécessaire mais bien évidemment pas suffisante pour que le monde soit idéal.

– Quel est le protocole de connexion/déconnexion ? Imaginez les différents cas de figures : une des deux machines émet un fichier, les deux émettent un fichier, une machine émet consécutivement deux fichiers pendant que l'autre n'en émet qu'un, initialement une seule machine émet, puis la deuxième émet, etc.
– Quelle structure de trames avez-vous choisie et pourquoi ?
– Quels sont les différents états possibles du protocole ? Dans chaque état, quels sont les différents événements qui peuvent survenir ? Quelles sont alors les réactions du protocole ?

4 La simulation de la couche physique et de l'application

Le fonctionnement de la couche liaison suppose l'existence d'une couche physique et d'une application cliente qui, *a priori*, sont des processus indépendants (figure 1).

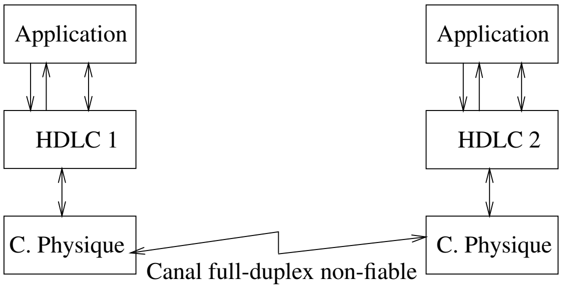

FIG. 1: Deux stations sont liées par un canal full-duplex non-fiable. Chaque station est composée de trois couches en communication. La couche liaison échange avec l'application cliente des informations de contrôle (*CONNECT.request, CONNECT.indication, DISCONNECT.request, DISCONNECT.indication*, etc.) et des données à émettre ou à recevoir (*DATA.request ou DATA.indication*).

En réalité, dans le cadre de ce projet, les couches serveurs et clients sont simulées par des fonctions. On n'aura donc pas affaire à une véritable couche application ni à une véritable couche physique, mais à des fonctions, qui, du point de vue de la couche liaison, émuleront l'application et la couche physique (figure 2).

Toutes ces fonctions vous sont fournies. Leur fonctionnement est décrit dans les sections suivantes. Autour d'une couche liaison presque inexistante, elles forment un programme $HDLC$. Ce programme est loin de mériter un tel nom dans son état actuel. La couche liaison se contente d'attendre l'arrivée d'un paquet ou d'une trame. Lorsqu'arrive un paquet, elle l'encapsule dans une trame, sans ajouter d'autres informations et remet la trame à la couche physique. Lorsqu'arrive une trame, elle en extrait le paquet et le remet à l'application. Ainsi, il n'y a ni détection d'erreur, ni retransmission, ni aucun des mécanismes que vous connaissez. Cependant, tant que le medium n'introduit pas d'erreurs, ce programme peut émettre et recevoir un nombre quelconque de fichiers.

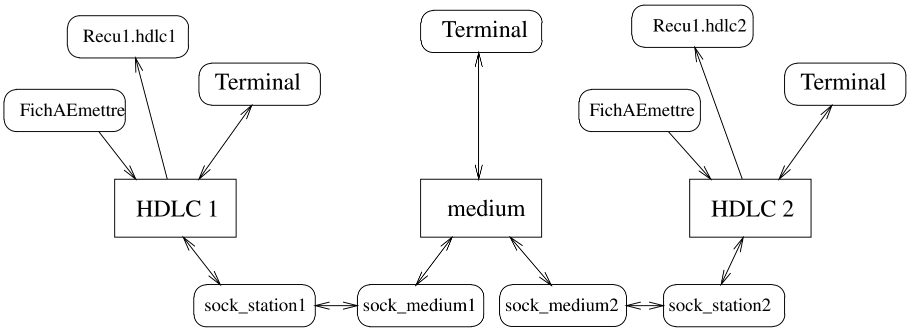

FIG. 2: La couche liaison communique avec les mêmes interfaces que sur la figure 1. Seulement de l'autre côté de cette interface, ce sont des fonctions et des descripteurs qui émulent la couche physique et l'application.

De plus, les données reçues ne sont pas regroupées en fichiers séparés, mais en un seul grand fichier. Votre protocole devra permettre d'enregistrer ces données dans des fichiers séparés. (Cette question sera vue plus en détail à la section 8.1)

5 Mode d'emploi

Le code de départ que nous vous proposons se trouve dans :

`~habibi/public/HDLC.d/CodeDepart.d`

Tous les fichiers sont accessibles en lecture. Vous pouvez donc les copier chez vous et les compiler (avec `make` ou `make linux`). Vous disposerez alors de deux programmes : *medium* et *HDLC*. Ces deux programmes communiquent au moyen de sockets de type `SOCK_DGRAM` sur le domaine `AF_UNIX`. Tels quels, ils ne peuvent fonctionner que s'ils sont lancés dans le même répertoire.

5.1 *medium*

Entrer simplement `medium`. Alors devrait apparaître le prompt

`MEDIUM :)`

Dès cet instant, le programme commence à écouter sur la socket `sock_medium1`, la socket `sock_medium2` et sur le terminal de contrôle. Toutes les données reçues sur la socket `sock_medium1` sont réémises telles quelles sur la socket `sock_medium2` à destination de la socket `sock_station2`. Réciproquement, toutes les données reçues sur la socket `sock_medium2` sont réémises telles quelles sur la socket `sock_medium1` à destination de la socket `sock_station1`. (Voir figure 2.)

Pendant ce temps, l'utilisateur peut interagir avec le programme et notamment demander l'introduction de différents types d'erreur. Entrer " ?" ou "help" affiche la courte liste des commandes possibles. Elles permettent essentiellement de connaître et de modifier le type d'erreur et la fréquence d'erreurs.

Les différents types d'erreur sont représentés par la variable `type`.

- `type==0` correspond à un canal fiable.
- `type==1` introduit un bit d'erreur dans chaque octet
- `type==2` introduit toujours des erreurs simples, dont la fréquence dépend de la variable `taux`. Pour `taux==0` le canal est fiable, pour `taux==1` le canal est identique à celui de `type==1`. Pour toutes les valeurs intermédiaires de `taux` on obtient des cas intermédiaires où les erreurs sont toujours simples mais n'ont pas lieu à tous les octets.
- `type==3` introduit une rafale de 8 bits faux dans chaque datagramme.
- `type==4` introduit toujours des rafales de 8 bits faux, mais leur fréquence dépend encore une fois de la variable `taux`. Pour `taux==0` le canal est fiable et pour `taux==1` tous les octets du datagramme sont des rafales d'erreur. Pour les valeurs intermédiaires de `taux`, on obtient des fréquences d'erreur intermédiaires.

Il est à noter que, tel quel, *medium* n'introduit pas de trame parasite, ni de rafale de longueur supérieure à 8.

5.2 *HDLC*

Entrer simplement `HDLC 1` ou `HDLC 2`. Alors devrait apparaître :

`HDLC1 :)`   ou   `HDLC2 :)`

L'argument (1 ou 2) est l'*identité* de la station. Il ne sert et ne doit servir qu'à nommer différemment les sockets et les fichiers de réception. Par ailleurs, le programme gérant les deux couches liaison est rigoureusement le même et doit le rester. La principale commande est "`envoi <fichier>`" Elle permet d'envoyer un fichier à la station distante. Une fois que l'émission est terminée on peut immédiatement envoyer un autre fichier et cela, sans la moindre corrélation avec toute réception de données ou d'absence de réception.

5.3 La session minimale

Pour vérifier que le programme qu'on vous fournit fonctionne :

– Ouvrir trois fenêtres ;
– lancer `medium` dans une fenêtre ;
– lancer `HDLC 1` dans une autre fenêtre ;
– lancer `HDLC 2` dans une troisième fenêtre ;
– sur cette troisième fenêtre entrer :

```
envoi LISEZ-MOI
```

– lorsque le programme indique `Émission terminée` vous pouvez vérifier que le fichier a été bien reçu en entrant dans l'une des fenêtres :

```
! diff Recu1.hdlc1 LISEZ-MOI
```

On pourra aussi faire des envois croisés.

Pour quitter le programme, on peut soit entrer "`quit`", (ou "`q`"). Le signal `SIGINT` (`ctrl c`) est traité par un handler et permet de sortir du programme après avoir fermé tous les fichiers, sockets et terminaux ouverts pendant le programme.

6 Vue d'ensemble du programme fourni

6.1 Votre domaine d'intervention

Le centre d'intérêt de ce projet est la couche liaison et son protocole. Les seuls fichiers consacrés à cette couche sont `Liaison.c` et `LiaisonToolbox.c`. Plus précisément, le protocole proprement dit est décrit dans la fonction

```
void ExecCoucheLiaison(int DataPhys, int CmdAppli);
```

décrite dans `Liaison.c`. Votre domaine de travail se limitera à ces deux modules et à ceux que vous aurez sans doute créés autour d'eux. Pour tester la robustesse de votre protocole, vous serez aussi sans doute amenés à modifier le module `Erreur.c` notamment pour augmenter la longueur des rafales d'erreur introduites par le canal, ainsi que pour émettre des trames parasites.

Ce sont les seuls modules que vous serez amenés à modifier, et sans doute les seuls dont vous devez comprendre le détail. Parmi les autres modules, certains contiennent des fonctions que vous pouvez utiliser. Mais cette utilisation ne nécessite pas la connaissance détaillée de leur fonctionnement interne. Les autres modules comme `SocketToolbox.c` ou `CtrlC_Propre.c` ne sont d'aucun intérêt pour le travail sur la couche liaison et peuvent être ignorés.

6.2 Les fonctions utiles

– Pour communiquer avec l'application et la couche physique, vous utiliserez les fonctions définies dans `Physique.c` et `Application.c`. Cf. les sections 8.1 et 8.2.
– Pour attendre un événement, on pourra utiliser la fonction `Attendre` qui est définie dans le module `Attente.c`. Cf. la section 9.1.
– Pour manipuler les paquets et les trames bit à bit, on pourra utiliser les fonctions qui sont définies dans le module `BitABitToolbox.c`. Cf. la section 9.2.
– Pour calculer les codes d'erreur et pour détecter les erreurs, on pourra utiliser les fonctions du module `Detection.c`. Cf. la section 9.3.

7 Les structures de données

Avant de commencer la description de ces fonctions, disons quelques mots sur les structures de données de la trame et du paquet. Dans l'état actuel du programme, le type `Paquet_t` est un caractère et le type `Trame_t` est également un caractère (cf. le fichier `define.h`). Il sera difficile de réaliser un protocole *HDLC* avec ces structures de données. Vous aurez donc à en définir d'autres, aussi bien pour la trame que pour le paquet. Les fonctions qui vous sont fournies fonctionnent quelques soient les types de données à transmettre. Il suffit simplement de les désigner par un pointeur et par une taille exprimée en octets. Par exemple, comme nous allons le voir, pour remettre une trame à la couche physique, nous disposons de la fonction :

```
int VersCouchePhysique(void *trame, int taille);
```

Si vous avez choisi une trame de type simple ou de type structure, alors l'appel se fera de la façon suivante :

```
typedef struct Trame_t
    {   char info;
        char controle;
    } Trame_t;            /* Ceci n'est qu'un exemple */
(...)
Trame_t trame;
VersCouchePhysique(&trame, sizeof(Trame_t));
/* L'important est le ``&trame'' */
```

Il en va de même pour toutes les fonctions qui vous sont fournies et qui prennent en paramètre un objet de type `void*` et sa taille. Par contre, si vous utilisez une trame de type pointeur ou tableau, alors l'appel sera légèrement différente :

```
typedef char Trame_t[4];   /* Ceci n'est qu'un exemple */
(...)
Trame_t trame;
VersCouchePhysique(trame, sizeof(Trame_t));
/* L'important est le ``trame'' */
```

Une macro `TYPE_POINTEUR` définie dans `define.h` distingue ces deux cas. Cette macro n'est utilisée que dans les fonctions `EncapsulerPaquet` et `DesEncapsulerPaquet` dans le fichier `LiaisonToolbox.c`. Mais il ne faudra pas oublier de la définir une fois que vous aurez choisi votre structure de données. Bien évidemment l'encapsulation et l'extraction des paquets dépendent également de votre structure de données et devront être redéfinies lorsque celle-ci aura été fixée.

8 Description des fonctions de l'interface

Pour respecter le principe des couches, le nombre des canaux de communications entre couches doit être très limité et la nature des informations de contrôle qui y circulent doit également être spécifique. Pour ce projet, l'interface à travers laquelle la couche liaison doit communiquer avec les autres couches vous est entièrement fournie. La couche liaison ne doit pas communiquer avec les autres couches autrement qu'au moyen de ces fonctions et de ces descripteurs. En particulier, toute variable globale entre la couche liaison et les autres couches est interdite.

Nous allons décrire le rôle de chacune des fonctions et chacun des descripteurs qui constituent l'interface de la couche liaison avec l'extérieur. Dans un premier temps, nous décrirons ces rôles comme celui d'une interface entre deux couches. (cf. la représentation de la figure 1) Mais en réalité, ces couches n'existent pas (cf. figure 2). C'est pourquoi, dans un deuxième temps nous dirons quelques mots sur ce que font réellement ces fonctions et la manière dont elles peuvent simuler la couche physique et l'application.

8.1 L'interface avec l'application

8.1.1 Situation de référence

Voici les éléments de l'interface entre la couche liaison et l'application cliente.

`DataAppli` et `CmdAppli`
Ce sont deux points de communication par lesquels passent respectivement les données et les informations de contrôle. Lorsque la couche liaison est en attente d'un événement elle ne fait qu'écouter ces points de communication (en même temps que l'homologue `DataPhys` de la couche physique (section 8.2.1)).

int OrigineApplication(void *paquet, int *taille);
Lorsque l'application a un paquet à transmettre, alors cette fonction permet à la couche liaison de récupérer ce paquet. Lorsque l'application n'a rien à transmettre, alors cette fonction bloque, jusqu'à ce que l'application ait effectivement quelque chose à transmettre.

Plus précisément, cette fonction lit *au plus* `*taille` octets dans le paquet émis par l'application et retourne, toujours à l'adresse `taille`, le nombre d'octets effectivement lus. Autrement dit, à l'appel de la fonction, le second paramètre `taille` doit pointer sur un nombre qui indique la taille *maximale* du message que l'on souhaite lire. Le premier paramètre `paquet` doit pointer sur un espace **alloué** dont la taille est au moins égale à `*taille`. Au retour de la fonction, le nombre pointé par `taille` indique le nombre d'octets effectivement lus.

Cette fonction retourne `CRASH` (c'est à dire -1) en cas de problème, elle retourne `EMISSION_FINIE` (c'est à dire 0) lorsque le fichier à transmettre a été entièrement transmis, et elle retourne `TOUT_BAIGNE` (c'est à dire 1) lorsque tout s'est bien passé.

int VersApplication(void *paquet, int taille);
La couche liaison utilise cette fonction pour délivrer un paquet à l'application. La variable `paquet` doit être un pointeur vers le paquet à émettre, quelque soit la structure de données sous-jacente. On suppose que l'application est toujours prête à accepter les paquets. La valeur de retour de la fonction est `CRASH` en cas de problème, et `TOUT_BAIGNE` si tout s'est bien passé.

int OrigineCmdAppli(int *commande, int *desc);
Lorsque l'application demande la transmission d'un nouveau fichier ou demande l'arrêt de toute transmission, alors l'information est véhiculée via `CmdAppli`. La fonction `OrigineCmdAppli` permet de lire cette information. Ainsi au retour de cette fonction, la variable `commande` peut avoir soit la valeur `QUIT`, soit la valeur `ENVOI`. Dans ce dernier cas, la valeur de la variable `desc` est celle du descripteur `DataAppli` qu'il faut utiliser pour les échanges de paquets de données.

La variable `commande` peut prendre d'autres valeurs que `ENVOI` et `QUIT`, mais la couche liaison ignore ces valeurs. Les valeurs de retour de cette fonction ont la même signification que pour les fonctions précédentes.

void VersCmdAppli();
Cette fonction permet à la couche liaison de transmettre des informations de contrôle à l'application. Dans notre cas, elle sert simplement à indiquer à l'application que la réception d'un des fichiers est terminée et que le fichier reçu peut être fermé. Ainsi, l'application saura que le prochain paquet qu'elle recevra devra être enregistré dans un nouveau fichier.

Dans l'état actuel du programme, cette fonction n'est presque jamais appelée. En effet, pour l'instant, la couche liaison n'a aucun moyen de savoir si une trame donnée provient du même fichier que la trame précédente ou non. C'est pourquoi toutes les trames sont enregistrées dans un seul et même fichier. Mais votre protocole devra permettre de séparer ces trames. Pour n fichiers émis par la station distante, le récepteur devra produire n fichiers reçus.

8.1.2 En réalité...

En réalité, `DataAppli` n'est rien d'autre que le descripteur (de type `int`) du fichier à émettre, et `CmdAppli` n'est rien d'autre que le descripteur du terminal de contrôle. Ces descripteurs permettent à la fonction `Attendre` (qui utilise `select`) d'attendre les commandes, les paquets ou les *timeout* (cf. la section 9.1). `OrigineApplication` ne communique pas avec une application, mais se contente de lire les données sur `DataAppli`, c'est-à-dire sur le fichier à émettre, et de rendre un paquet de taille voulue. De même, `VersApplication` ne fait qu'écrire le paquet reçu dans le fichier de réception. La fonction `VersCmdAppli()` se contente de fermer le fichier de réception et de se préparer à en ouvrir un autre à la réception suivante.

Enfin, la fonction `OrigineCmdAppli` ne fait que transmettre certaines commandes venant du terminal de contrôle à la couche liaison, éventuellement en indiquant le descripteur du nouveau fichier à transmettre.

8.2 L'interface avec la couche physique

8.2.1 Situation de référence

Voici les éléments de l'interface entre la couche liaison et la couche physique.

Le descripteur `DataPhys`
Il s'agit d'un point de communication par lequel passent toutes les trames que la couche liaison cherche à transmettre sur le canal ou à lire sur le canal. Lorsque la couche liaison est en attente de l'arrivée d'une trame elle ne fait qu'écouter ce point de communication.

`int OrigineCouchePhysique(void *trame, int *taille);`
Lorsque la couche physique a reçu une trame, alors cette fonction permet à la couche liaison de lire cette trame. Lorsque la couche physique n'a reçu aucune trame, alors cette fonction bloque, jusqu'à ce que la couche physique reçoive effectivement une trame.

Plus précisément, cette fonction lit *au plus* `*taille` octets dans la trame reçue par la couche physique, et retourne, toujours à l'adresse `taille`, le nombre d'octets effectivement lus. Autrement dit, à l'appel de la fonction, le second paramètre `taille` doit pointer sur un nombre qui indique la taille *maximale* du message que l'on souhaite lire. Le premier paramètre `trame` doit pointer sur un espace **alloué** dont la taille est au moins égale à `*taille`. Au retour de la fonction, le nombre pointé par `taille` indique le nombre d'octets effectivement lus. Les valeurs retournées par cette fonction ont la même signification que pour les fonctions précédentes.

`int VersCouchePhysique(void *trame, int taille);`
Lorsque la couche liaison a une trame à transmettre, elle utilise la fonction `VersCouchePhysique` en lui donnant un pointeur sur la trame à transmettre. Les valeurs de retour de cette fonction ont la même signification que pour les fonctions précédentes.

8.2.2 En réalité ...

En réalité, le descripteur `DataPhys` n'est que le descripteur de la socket qui permet aux stations de communiquer. Les fonctions `OrigineCouchePhysique` et `VersCouchePhysique` ne font respectivement que lire et écrire sur cette socket.

9 Quelques outils

9.1 La fonction `Attendre`

Les fonctions `Origine...` sont bloquantes. Donc on ne peut pas utiliser ces fonctions telles quelles. Il faut les appeler lorsque l'on sait qu'il y a effectivement des données (paquet ou trame) à lire. Pour cela, on utilise la fonction :

```
int Attendre(int fd1, int fd2, int fd3, float duree);
```

où `fd1`, `fd2` et `fd3` sont des descripteurs de fichiers, de tubes, de sockets ou de terminaux.

Cette fonction reste bloquée jusqu'à ce qu'il y ait quelque chose à lire sur au moins l'un des trois objets représentés par ces trois descripteurs, ou jusqu'à ce que la durée `duree` (exprimée en secondes) se soit écoulée *(timeout)*. Cette fonction utilise la primitive `select`.

Les valeurs de retour de cette fonction sont des constantes définies par macro.

- s'il y a quelque chose à lire sur l'objet décrit par `fd1`, (et seulement sur cet objet) alors la fonction rend la constante `YaALireSur1__`;
- s'il y a à lire sur le second descripteur, et seulement sur ce descripteur, la valeur rendue sera `YaALireSur_2_`;
- s'il y a à lire en même temps sur le premier descripteur et le deuxième descripteur, alors la valeur rendue sera `YaALireSur12_`;
- s'il y a à lire sur les trois descripteurs, alors la valeur rendue sera `YaALireSur123`;
- etc ;

– enfin si aucun des objets n'est lisible au bout d'un temps `duree`, alors la valeur rendue est `TIMEOUT`.

Si on ne désire pas attendre des événements sur 3 descripteurs, mais sur un seul, par exemple `fd`, on peut faire l'appel suivant :

```
int evenement;
evenement = Attendre(fd, RIEN, RIEN, 5.0);
```

Cet appel attend pendant au plus 5 secondes que quelque chose soit lisible sur `fd`. Les seuls événements (i.e. les seuls valeurs de retour) possibles sont `YaALireSur1__` et `TIMEOUT`. L'appel :

```
Attendre(RIEN, RIEN, RIEN, 5.0);
```

est l'équivalent de `sleep(5);`

Enfin, si on ne cherche pas à armer la temporisation, alors la valeur du paramètre `duree` doit être nulle :

```
evenement = Attendre(fd1, fd2, RIEN, 0.0);
```

Dans ce cas, quelque soit la durée d'attente, la fonction ne se débloquera que si `fd1` ou `fd2` (ou les deux) deviennent lisibles.

9.2 Les fonctions `BitABit`

Un certain nombre de fonctions vous permettent de manipuler les nombres bit à bit. Les objets qu'il est possible de manipuler ne se limitent plus aux caractères et aux entiers courts, mais peuvent être tout à fait quelconques, pourvu qu'on puisse les désigner par un pointeur et par une taille mémoire exprimée en octets.

Ainsi, que l'on représente les paquets et les trames par des structures, des tableaux, ou des types simples, ces fonctions vous permettent d'afficher leur contenu bit par bit, de connaître la valeur du bit numéro n, et de changer la valeur de ce bit , etc.

ATTENTION ! ! !
Le bit des unités n'est pas le bit 1, mais le bit 0. Ainsi les huit bits d'un caractère sont numérotés de 0 à 7.
Un polynôme de degré 16 a des bits allant de 0 à 16.

9.3 Le codage et le décodage

Le polynôme générateur que vous utiliserez est un polynôme normalisé. Il s'agit du polynôme $X^{16}+X^{12}+X^5+1$. Ce nombre est désigné par la macro `GENERATEUR` définie dans `Detection.h`.

Soit une information à coder. Nous la représentons sous forme polynomiale $I(X)$. Dans notre cas, cette information doit renfermer non seulement le paquet de données, mais aussi tous les autres champs de la trame *HDLC* à l'exception du code d'erreur. Pour coder $I(X)$, il faut d'abord le multiplier par X^N où N est le degré du polynôme générateur. Si votre structure de données n'a pas prévu de place pour ces 0 supplémentaires, alors vous pouvez utiliser la fonction :

```
void* AllongerDeDeuxOctets(void *objet, int taille);
```

qui, étant donné un objet `objet` avec n bits ($n = 8\times$ `taille`), rend un objet avec $n + 16$ bits. Les bits de poids fort ont le même contenu que ceux de `objet` et les 16 bits de poids faible sont nuls.

ATTENTION ! ! ! Cette fonction libère l'espace pointé par `objet`.

Une fois que vous disposez de $X^N \cdot I(X)$, il ne reste plus qu'à le diviser par le polynôme générateur et en retirer le reste. Ceci peut être effectué par la fonction :

```
short ResteDivisionModulo2(void *dividende, int taille, int diviseur);
```

qui rend un entier court qui représente le reste de cette division.

Pour toute question ou réclamation sur ce code, envoyer un courrier électronique à
`habibi@dpt-info.u-strasbg.fr`