PROJET C

Commutateur

Crosnier Michael, Dakkak Miryam, Manteau Julien
– mcrosnier, mdakkak, jmanteau –
Groupe 14, TR1/2

Le but de ce projet est de « simuler le travail d'un élément réseau imaginaire ». Il doit pouvoir, à partir de trames (L-PDU), reconstituer une N-PDU, qui passera dans la couche réseau pour y être commutée. Ce paquet reviendra alors à la couche liaison pour y être de nouveau décomposé en trames, et envoyé.

Vu la complexité du problème posé, nous avons fait le choix de subdiviser le programme principal en divers modules, que nous allons définir dans ce qui suit. Il suffira alors de les appeler pour faire fonctionner le programme principal: « commutateur ».

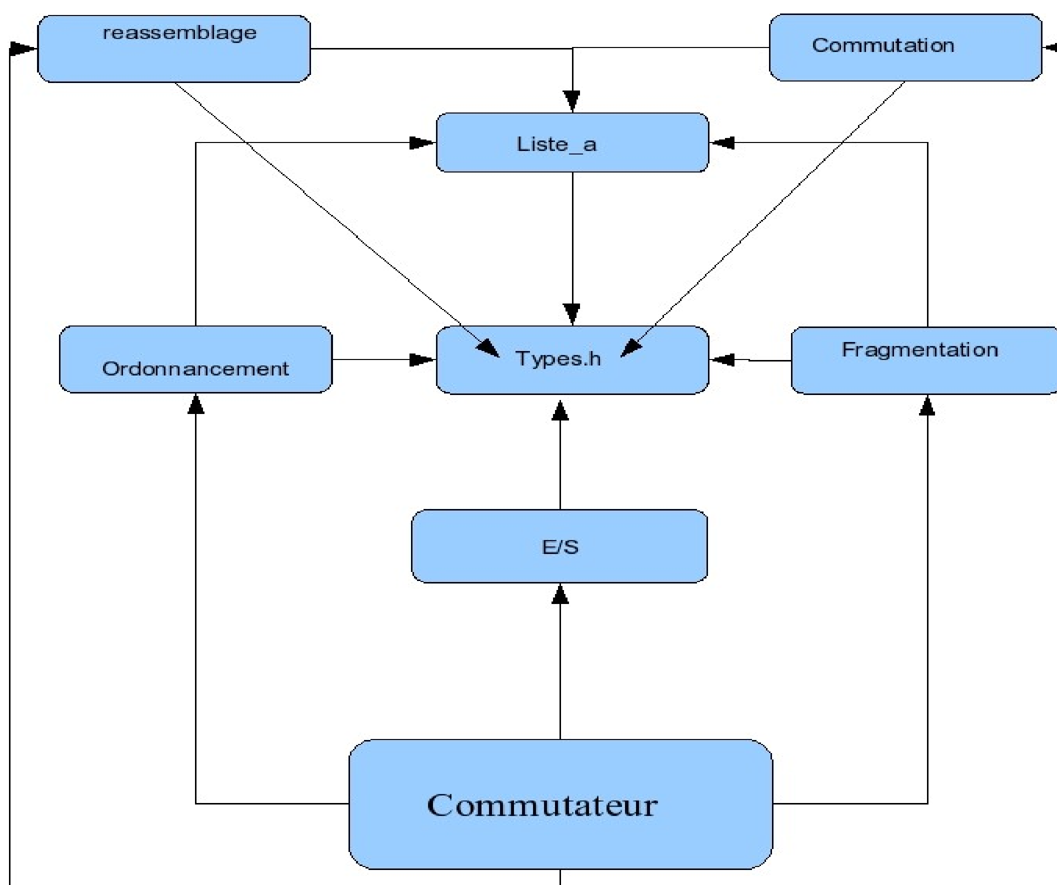
Nous verrons également les difficultés auxquelles nous avons été confrontés ainsi que les projets d'améliorations pour de futures programmations.

Table des matières

I. L'architecture adoptée:....................2
I.1. L'arcitecture des modules....................2
I.2. Les strucures de données....................2
II. Les modules 4
II.1. Le module Entrées / Sorties : « ES »....................4
II.2. Le module « Reassemblage »:....................5
II.3. Le module « Commutation »....................5
II.4. Le module « Fragmentation »....................6
II.5. Le module « Ordonnanceur »....................6
II.6. Le module « Liste_a »....................7
III. Le programme principal....................8
IV. Exemple d'algorithme (Dans le module ES)....................8
V. Difficultés rencontrées et choix réalisés....................10
V.1. Difficultés liées à la compréhension du sujet....................10
V.2. Problèmes dans le module « Lecture/Ecriture »....................10
V.3. Les tests....................10
V.4. Le travail en trinôme....................10
VI. Conclusion....................11
VI.1. Améliorations possibles?....................11
VI.2. Enfin....................11

I. L'architecture adoptée:

I.1. L'architecture des modules



L'utilité de chacun de ces modules sera définie par la suite

I.2. Les strucures de données

Les formats:

Les formats suivants on été définis afin de simplifier les implantations

```
typedef char format_int_8;
typedef unsigned short int format_int_16;
typedef unsigned int format_int_32;
```

Les types

Nous avons, après avoir examiné soigneusement ce qui nous était demandé et nous être mis d’accord sur la compréhension du sujet, décidé de créer un module qui contiendrait les types les plus utiles et qui définirait les structures de données suivantes:

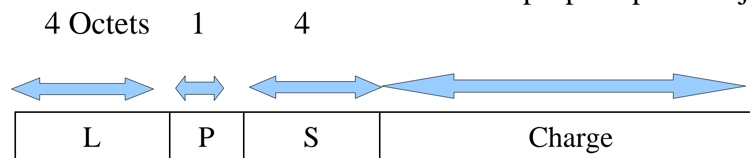
Le Type N-PDU:

Une N-PDU est le message contenu dans la couche réseau. Elle est également appelé « paquet » et dans notre cas, constitue le message directement lié à la commutation (qui est d'ailleurs la principale fonction de la couche 3).

```
typedef struct NPDU{
    format_int_32 L; // Label
    format_int_8 P;  // Priorité
    format_int_16 S; // Taille de la charge utile
    Charge charge;   // Charge utile de la N-PDU
} NPDU;
```

Cette définition est en accord avec le format proposé par le sujet :

4 Octets 1 4



Le Type L-PDU:

Parallèlement à la NPDU, la LPDU est le message relatif à la deuxième couche du réseau, qui est la couche liaison. Une LPDU résulte du découpage d'une NPDU en fragments, auxquels il faut rajouter un en-tête qui permet de les identifier.

Comme nous ne pouvons pas travailler en temps réel, la couche liaison ne peut pas décider elle-même du moment où elle a pour rôle de réassembler les L-PDU par rapport à celui où elle doit fragmenter et envoyer les L-PDU.

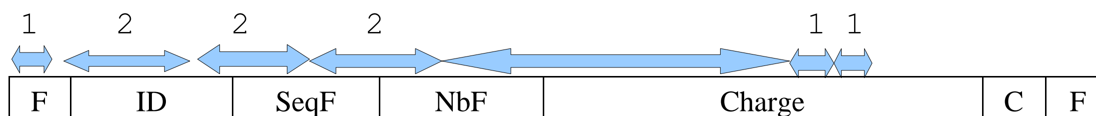
C'est la raison pour laquelle nous sont fournis des fichiers qui servent à piloter les entrées et les sorties du commutateur : chaque mot IN contenu dans ce fichier sera suivie d'une trame à lire, et chaque mot OUT signifiera qu'il faut émettre une trame.

Nous définissons ce type de la manière suivante :

```
typedef struct LPDU{
     format_int_8 F;     // Fanion
     format_int_16 ID  ;// Identifiant
     format_int_16 SeqF;// numéro de séquence i
     format_int_16 NbF; // nb total de fragments n
     Charge charge;     // la charge de la L-PDU
     format_int_8 C;    // Code détecteur d'erreur
} LPDU;
```

La taille de la charge pouvant varier, on alloue celle-ci dynamiquement ainsi, le type « Charge » est le suivant:

```
Typedef struct Charge {
      char* donnees;
      size_t taille;
} Charge;
```



Le Type Options_Commandes:

Nous avons défini ce type dans le but de simplifier les opérations de codage (en particulier dans le programme principal). En effet, lorsqu'il s'agit par exemple de fragmenter une NPDU en plusieurs LPDU, l'utilisateur doit nous indiquer une taille de fragmentation, qui est donnée par la ligne de commande.

Elle devra par ailleurs prendre en entrée plusieurs autres options:

```
typedef struct options_commande{
     char* input;
     char* output;
     char* table_file;
     int input_size;
     int output_size;
     int link_number;
 } options_commande;
```

II. Les modules

II.1. Le module Entrées / Sorties : « ES »

Ce module gère la lecture/l'écriture dans les fichiers, l'utilisation de la ligne de commande, et enfin le formatage en L-PDU des lignes « in » contenues dans le fichier fourni en entrée par l'utilisateur.

Ce module proposera les sous-programmes suivants:

- `bool ecriture_LPDU(unsigned char* output_file,int lien,char* output_prefix, size_t taille)`
  Ecrit les trames dans un fichier mais écrit une seule ligne à la fois
  Renvoie true si ça c'est bien passé
- `char* nomme_sortie(int lien,char* output_prefix)`
  Donne le nom du fichier de sortie en entrant le lien et le prefixe de sortie
- `LPDU* formatage_LPDU(unsigned char* lpdu_brute, int taille_lpdu);`
  Formate l'entrée de façon à renvoyer une LPDU à partir de unsigned char et de i le nombre de char envoyés*
- `Cellule_Entree lire_ligne(FILE *flux)`
  *Lit les trames dans un fichier mais lit une seule ligne (IN, OUT , LPDU) à la fois (en fait * renvoie la prochaine ligne du flux)*
- `void options_defaut(Positionnement pos, options_ com- mande* options)`
  Remplit l'entrée options_commande en fonction des valeurs par défaut non rentrées par l'utilisateur

- `void entree_terminal(int argc,char ** argv, options_ commande* options);`
  Gère l'entrée par la ligne de commande, exemple:
  commutateur -i mon_entree -p ma_sortie -t table_commut -fi size_frag_in -fo size_frag_out -l nb_lien
  (i,p,t,fi,fo): arguments
  (mon_entree,ma_sortie,table_commut...): options

II.2. Le module « Reassemblage »:

Le module Réassemblage permet de recréer les NPDU à partir des L-PDU fournies par le module E/S. Il aura fallu vérifier le contenu des trames grâce au CRC, qui est ici le « ou exclusif » . Une fois reconstitués, les paquets seront transmis au commutateur.

- `NPDU assembler_paquet (liste_a_t liste_a, format_int_16 ID,options_commande options);`
  Cette fonction reconstitue la N-PDU à partir de la liste d'attente créée précédemment. Celle ci fournit des charges chainées et, pour passer de l'une à l'autre, il suffit alors d'enlever l'élément au sommet, on accèdera alors à la suivante.

- `bool NPDU_prete_reassemblage (liste_a_t* liste_a, format_int_16 NbF, format_int_16 ID);`
  Renvoie TRUE si une NPDU est prete à etre émise de la liste d'attente à partir du NbF

- `void remplir_liste_a(liste_a_t* liste_a, LPDU l)`
  Ce sous-programme remplit la liste d'attente liste_a avec les charges corrigées des trames fournies par le fichier en entrée

II.3. Le module « Commutation »

Le but de ce module est de lire la table de commutation pour rediriger le paquets. En recevant celui-ci, on substitue le label d'entrée par le label de sortie et on envoie le nouveau paquet vers lel ien de sortie indiqué par la table.

Pour commencer, on définit les types suivants:

- `typedef format_int_32 label_t;`
- ```
  typedef struct {
      size_t taille;
      tab_label_t labels;
  }tab_com_t ;
  ```
- `typedef format_int_32 lien_t;`
- `bool creer_tableau_commu (tab_com_t* tab, options_commande option);`
  Crée un tableau d'entier de la table de commutation. Si l'allocation
  ```

mémoire du tableau n'a pas réussi, le sous-programme renvoie un booléen faux. Il s'agit ainsi de chercher le nombre de lignes de la table de commutation, d'allouer la memoire et de calculer la taille du tableau label pour enfin le remplir, en pensant à noter les éventuelles erreurs de lecture des valeurs de la table de commutation.

- `bool liens_sont_valides (tab_com_t tab_comm, options_commande option);`
  Vérifie la validité des numéros de liens dans la table de commutation.
- `bool commuter (NPDU * paquet,lien_t* lien_sortie, tab_com_t tab_comm);`
  C*hange le label en fonction de tab_comm et trouve le lien de sortie du paquet. Si le label d'entrée du paquet n'est pas dans la table de commutation, le sous-programme renvoie faux.*
- `static int indice_label_entre (label_t label_entre, tab_com_t tab_comm);`
  Renvoie l'indice du label d'entrée dans la table de commutation
  Necessite : (label_entre_present (label_entre,tab_comm)

II.4. Le module « Fragmentation »

Ce module a pour objectif de créer les LPDU issues de la fragmentation d'une NPDU et leur attribue un ID.

- `size_t nbr_banaliseur_a_ajouter(unsigned char * caras, size_t taille)`
  Renvoie le nombre de banaliseur à ajouter dans un tableau de caractere de taille « taille »
- *`format_int_8 crc_lpdu (lpdu_chaine_cara_t lpdu,size_t taille_lpdu);`*
  Renvoie le CRC de la lpdu sans le fanion
- `bool ajouter_banaliseur (lpdu_chaine_cara_t *lpdu, size_t taille_lpdu)`
  A*joute les banaliseur dans la lpdu*
- `bool fragmenter_NPDU (NPDU npdu,liste_a_t * liste_a, options_commande option)`
  Dans ce sous-programme, on détermine la taille de chaque fragment, puis on crée l'une après l'autre les Lpdu. On reconstitue d'abord la première, qui contient en fait l'en-tête de la NPDU puis les autres, dans un second temps. Chaque fois qu'une LPDU est créée, elle est ajoutée à laliste d'attente

II.5. Le module « Ordonnanceur »

L'objectif de ce module est de gérer la politique de *Weighted Round Robin.*
En effet, il détermine le cp (nombre de messages à envoyer) et écrit en sortie la prochaine trame à envoyer.

- `void initialiser_ordonnanceur (Ordonnanceur* ord);`
  initialise le tableau ordonnanceur
- `int credit (format_int_8 p);`
  détermine la valeur du cp qui est le nombre de messages à envoyer
- `int prochain_cp (Ordonnanceur* ord, int cp_precedent, int num_lien)`
  détermine le prochain cp qui devra être envoyé
- `void liberer_ordonnanceur(Ordonnanceur* ord);`
  Détruit l'ordonnanceur

- `void ordonnancer(Ordonnanceur ordonnanceur,unsigned int lien,size_t* taille_a_ecrire,char* a_ecrire);`
  Met dans a_ecrire la prochaine trame à écrire en sortie et donne sa taille

II.6. Le module « Liste_a »

Il gère toutes les opérations à effectuer sur le type « liste d'attente ». Le module « réassemblage » s'en sert notamment pour stocker dans l'ordre les charges des L-PDU, et ainsi recréer facilement une N-PDU. D'autre part, il permet également d'établir un lien entre les deux modules « fragmentation » et « ordonnanceur ».

Types utilisés:

- `typedef format_int_16   cle_t;`
-
```
typedef struct {
    Ens debut;
    cle_t cle;
    format_int_16 nbr_element;
    format_int_16 nbr_elem_prevu;
} liste_t;
```
- `typedef liste_t * listes_t;`
-
```
typedef struct {
 size_t taille;
 size_t nbr_cle;
 listes_t listes;
}liste_a_t;  /*  Le type de liste_a est un tableau de Liste */
```

Sous-programmes:

- `bool init_liste_a ( liste_a_t * liste_a);`
  Crée une liste chainée associative de taille_initial
- `bool ajouter_element (liste_a_t * liste_a, element_t e, cle_t cle, format_int_16 nbr_elem_prevu);`
  Ajoute un élément au sommet de la liste qui a pour clé « cle ».
- `void detruire_liste_a ( liste_a_t * liste_a);`
  Détruire toute la liste
- `void detruire_liste ( liste_a_t *liste_a, cle_t cle);`
- `bool enlever_sommet (liste_a_t *liste_a, cle_t cle);`
- `element_t sommet (liste_a_t liste_a, cle_t cle);`

D'autres sous-programmes existent dans ce module, ils proposent diverses opérations utiles au travail sur la liste d'attente. Il ne nous a pas semblé intéressant de tous les énumérer.

III. Le programme principal

Le programme principal regroupe, dans l'ordre, chacun des sous-programmes conçus dans les modules cités précédemment. En effet, C'est le programme qui contient le « main » qui permet d'atteindre l'objectif du projet.
On passe ainsi, dans l'ordre, par les étapes suivantes:

```
on regarde les entrées utilisateur du terminal
on ouvre le fichier d'entrée

Initialisation de l'ordonnanceur
Initialisation de la table de commutation
Initialisation de la liste d'attente associative avant réassemblage

tant que le fichier d'entrée n'est pas fini, faire
|       lecture l'entrée dans le fichier
|       Si une trame est émise (choix == IN) alors
|       |       on remplit la liste d'attente avec les LPDUs
|       |       Si une NPDU est prête à etre réassemblée alors
|       |       |       On réassemble la NPDU
|       |       |       On la commute
|       |       |       On la fragmente
|       |       Fsi
|       Sinon si  on demande une trame en sortie (choix == OUT) alors
|       |       On selectionne la prochaine trame à écrire en sortie
|       |       Si il y en a effectivement une alors
|       |       |       L'écrire
|       |       Fsi
|       Sinon si la trame est erronée (choix == CRC) alors
|       |       On rejette la trame
|        Fsi
FTQ
```

IV. Exemple d'algorithme (Dans le module ES)

Ce sous-programme lit les trames dans un fichier mais lit une seule ligne (IN, OUT , LPDU) à la fois (en fait renvoie la prochaine ligne du flux) :

```
Cellule_Entree lire_ligne(FILE *flux){

unsigned char* ligne= NULL ; // la ligne à renvoyer
unsigned char* tmp=NULL ; // le buffer
unsigned char c;
char choix[4];
unsigned char entree[4];
int lien;
int i= 0;
int num_lu;
Cellule_Entree ligne_formatee;
if (flux != NULL ){
```

```
 num_lu=fread(entree,sizeof(char),3,flux);
 // lecture dans une chaine buffer
 entree[3]='\0'; // on finit la chaine pour lire dedans
 sscanf(entree,"%s",choix); // lecture dans la chaine
 if ( strcmp(choix,"IN")==0 ) // c'est un IN!{
    // on enregistre le in
    ligne_formatee.choix= IN;
    // un IN est suivi d'une trame
    do {
     // capture de la trame jusqu'à renconter un fanion (\n)
    // Capture du caractère suivant
    c= getc(flux);
    // Capture de tout l'entete puis jusqu'à un fanion
    if ( (c != fanion)  || (i<7) ){
          // si banaliseur on l'enlève et on passe au suivant
          if ( c==banaliseur){
            tmp= realloc(tmp,sizeof(unsigned char));
            if (tmp != NULL){
                c= getc(flux);
                ligne= tmp;
                ligne[i]= c;
                i++;
                c = 'a'; // on trompe le test de fin
            } else {
                printf("Erreur d'allocation mémoire\n");
            }
          }else // caractère normal{
            tmp= realloc(tmp,sizeof(unsigned char));
            if (tmp != NULL){
                ligne= tmp;
                ligne[i]= c;
                i++;
            }else {
                printf("Erreur d'allocation mémoire\n");
            }
          }
      }
    }
    while ( (c != fanion) || (i<7)  );
    // on formate notre LPDU
   ligne_formatee.lpdu= formatage_LPDU(ligne,i);
    // si on nous retourne NULL: erreur de CRC
   if ( ligne_formatee.lpdu == NULL)  {
      ligne_formatee.choix= CRC;
   }
}else if ( strcmp(choix,"OUT")==0 ) // c'est un OUT!{
    ligne_formatee.choix= OUT;
    fscanf(flux," %d",&lien);
    // lecture dans le flux du numéro de lien
    ligne_formatee.lien =lien;
    c = getc(flux);
    // avance pour arriver au prochain IN ou OUT
}else if ( strcmp(choix," ") ){
    // c'est la fin du fichier!
```

```
            ligne_formatee.choix= EoF;
            printf("\n Fin du fichier \n");
        }
    }
    free(tmp);
    tmp=NULL;
    return ligne_formatee;
}
```

V. Difficultés rencontrées et choix réalisés

V.1. Difficultés liées à la compréhension du sujet

La compréhension du sujet constitue en elle-même une difficulté majeure. En effet, bien qu'à première vue, il semble compréhensible grâce aux acquis du cours de réseaux, lorsque l'on s'y intéresse de plus près, le problème nous invite naturellement à nous poser de nombreuses questions.

Il a donc fallu se concerter, pour mettre en commun nos informations et surtout pour confronter nos interprétations du sujet. Si celles-ci divergeaient, il fallait poser le problème et en trouver la solution.

Il fallait alors, établir un lien entre le codage, qui nous semble à première vue abstrait, et les applications concrètes auxquelles nous avons l'habitude d'être confrontés en cours de réseaux.

V.2. Problèmes dans le module « Lecture/Ecriture »

Certains problèmes sont apparus en particulier dans ce module. Ainsi par exemple, la lecture de fichiers peut poser quelques problèmes: il faut bien pouvoir décomposer les différentes LPDU, pouvoir reconnaître efficacement les in et out. Cela suscite quelques soucis d'odre technique car il faut alors trouver les fonctions adaptées au problème. En effet, on ne lit pas, ici, tout le fichier caractère par caractère mais il nous arrive parfois de le traiter par chaine (le in et le out sont par exemple considérés comme des chaines alors que les trames sont, pour nous des « unsigned char*).

Il fallait, par ailleurs se mettre d'accord sur une structure de données réutilisable au niveau du réassemblage, et qui permettrait donc de retrouver la trame lue dans le fichier

Enfin, ce module s'occupe de la ligne de commande, ce qui implique forcément la prise en compte de mesures de sécurité destinées à éviter tout incident causé par une mauvaise manipulation de l'utilisateur.

V.3. Les tests

Une autre difficulté est, ici, imposée par l'architecture même du projet. En effet, afin de vérifier le bon fonctionnement de nos modules, nous devions les tester régulièrement et séparément. Il fallait par ailleurs s'assurer de l'utilité de chaque ligne de code, et donc du fait que le programme se sert au moins une fois de chacune d'entre elles.

V.4. Le travail en trinôme

Enfin, il ne faut pas non plus oublier que ce travail est un projet à mener en trinôme. Toute la difficulté réside donc dans la communication, particulièrement dans mises à jours des fichiers. Il faut donc constamment travailler en réseau et mettre à jour la moindre de nos modifications, car les programmes étant liés, une transformation dans l'un des modules engendre bien souvent une modification, en parallèle, sur d'autres.

C'est d'ailleurs la raison pour laquelle nous avons choisi de créer un module « types.h », comme nous avons pu le voir précédemment, qui nous empêchait de modifier les définitions des types les plus importants, celles utilisées par tous les modules.

VI. Conclusion

VI.1. Améliorations possibles?

Notre projet n'étant évidemment pas parfait, certaines améliorations sont possibles. Nous remarquons ainsi par exemple que tester le module « reassemblage » n'est pas réellement possible tant que les modules « ES » ou « fragmentation » ne sont pas finis.

D'autre part, n'ayant pas dès le départ travaillé en réseau, nous avons perdu du temps, au début, à mettre en commun nos informations. Cela demandait effectivement de nombreux réglages (droit d'écriture, de lecture etc...).

VI.2. Enfin...

Ce projet nous a permis en particulier, d'établir un lien entre les réseaux et l'informatique. Il permet notamment de nous pencher sur l'implémentation basique d' un commutateur, élément clé en réseaux. L'écriture de modules apporte aussi beaucoup d'informations et d'amélioration en ce qui concerne la programmation. En effet, il s'agissait ici de créer des modules qui ont chacun une utilisation particulière. La difficulté consistait donc à décider quels choix de modules il fallait faire. Mais elle nous a aussi appris à prendre des décisions quant aux choix d'implantation.

Le travail en trinôme, enfin, qui est une nouveauté, nous a un peu plus envoyés vers le monde de l'entreprise, où les projets sont habituellement menés par petits groupes.

ANNEXE 1

<table>
<tr><th></th><th>Michael</th><th>Miryam</th><th>Julien</th></tr>
<tr><td>Compréhension
Conception</td><td colspan="3">Les deux premières séances de TP de deux heures ont été consacrées à la compréhension du sujet et à sa conception.
Ce travail a été fait ensemble</td></tr>
<tr><td>Modules traités</td><td>Liste_a
ragmentation
commutation</td><td>Réassemblage
Ordonnancement</td><td>Types.h
E/S</td></tr>
<tr><td>Programmation</td><td colspan="3">Nous avons travaillé sur la programmation en même temps, sur toutes les séances de TP (une trentaine d'heures)
Auxquelles il faut rajouter environ une quinzaine en dehors de ces séances (chacun programmait ses modules)</td></tr>
<tr><td>Mise au point</td><td colspan="3" rowspan="2">Ces deux parties nous ont pris une dizaine d'heures chacun
Elles ont été effectuées au même moment</td></tr>
<tr><td>Préparation de la présentation</td></tr>
</table>

Nous pouvons par ailleurs noter que finalement, nous avons pu gérer notre temps, malgré, il est vrai, quelques difficultés à la fin.