Le Mans Université

175UD04 Codage de l'information

Compte Rendu de TP

Rodrigue Meunier - Nicolas Lemoult - Mathis Morisseau

10 décembre 2021

Table des matières

1 Introduction

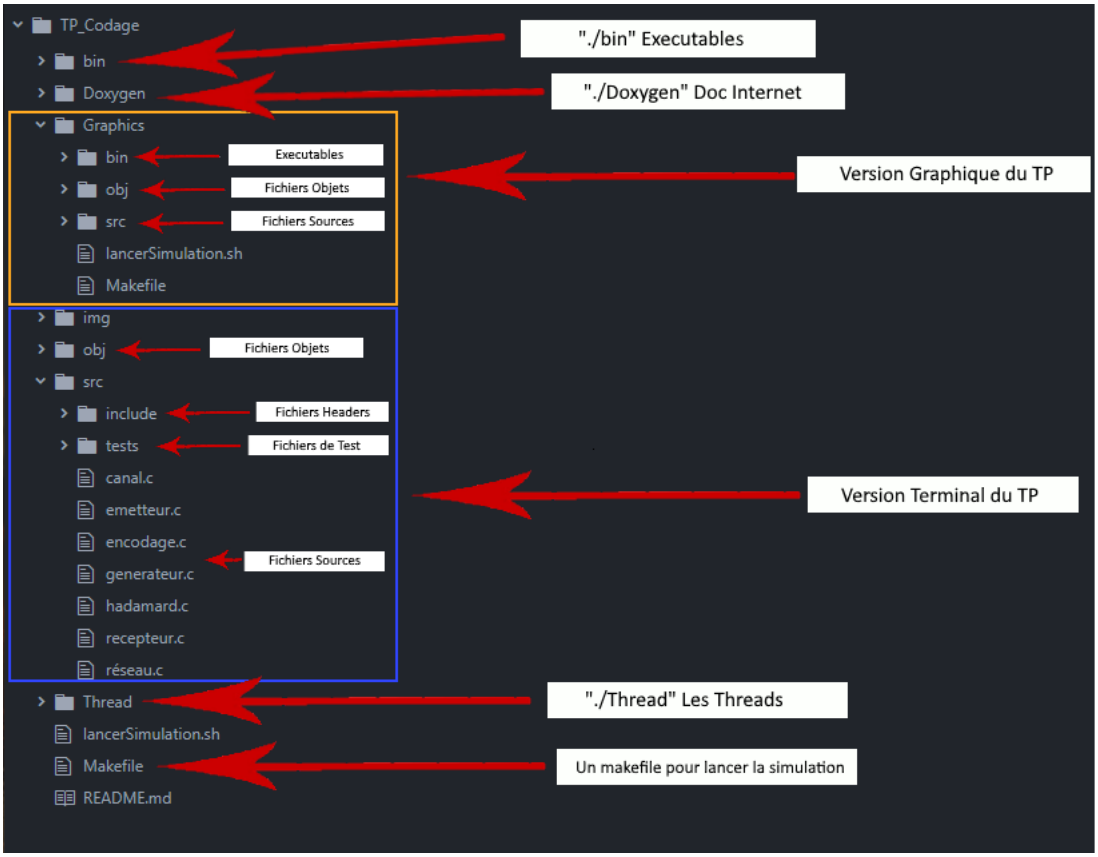
L'objectif du tp est de réaliser, par le biais de plusieurs séances et de plusieurs sujets de tp, un mini projet.
On peut surtout voir l'objectif principal comme étant la mise en pratique des notions vues en cours, permettant une meilleure approche dans le monde réel que l'aspect théorique du cours.
Ce mini projet a pour objet la simulation d'un environnement de 3G, il y a un *Emetteur* qui encode des messages, ceux-ci sont transmit à une entité *Canal* qui ajoute un bruit puis sont enfin envoyé au *Recepteur*.
Pour réaliser ce projet nous avons eu 3 séances de 3h avec pour chacune un sujet de TP supplémentaire. Chaque sujet est composé de quelques questions dans le but de nous indiquer ce qu'il faut rajouter dans le projet.
Les nombreux problèmes à résoudres lors de ce tp sont certainement de comprendre parfaitement les notions pour les retranscrire en *C*.En effet, nous allons manipulé une quantité assez conséquente de données et il est pas improbable de tomber sur quelques problèmes mémoire de types *Segmentation fault*.
Ce tp bien que peu complexe va être structuré avec des fichiers de tests permettant de tester individuellement chaque module de ce projet (Encodage - Fonctions de Hadamard - Fonction de génération - Fonction de réseau(*sockets*))

2 Déroulement du TP

Chaque sujet a été décomposé en plusieurs parties pour que chacun ait quelque chose à faire.
Nous ne voulions pas qu'il y ait qu'une seule personne sur chaque partie. L'objectif des TP est de mettre en pratique ce que l'on a vu en cours, il faut donc toucher à tout !

2.1 Organisation des fichiers

Avant toute chose il est important de bien comprendre comment sont agencés les fichiers pour une meilleure navigation dans notre abrorescence.



2.2 Comprehension

2.2.1 Vue d'ensemble

Nous avons déterminé qu'il faudrait 3 entités distinctes pour notre simulation à savoir : *Emetteur - Canal - Recepteur*.
Chaque programme est lancé quasi indépendamment des autres, petite exception : canal.
En effet *canal* agit comme un serveur car il doit recevoir les informations de l'*emetteur*

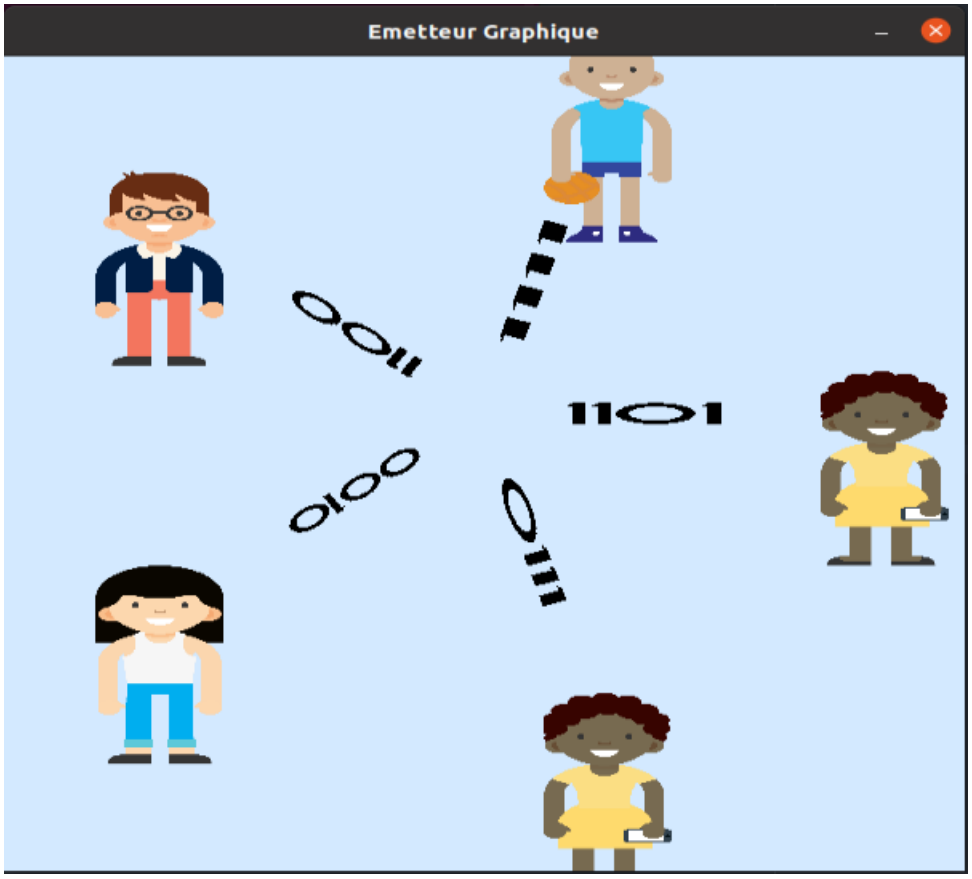
et les envoyer au *recepteur*.
Pour la réalisation des fonctions principales d'encodage et de Hadamard nous avons besoin de connaitre la **taille des messages** et également le **nombre d'utilisateur**.
C'est pour cela que nous avons besoin de lancer en premier *canal* qui va demander à l'utilisateur les valeurs de ces variables et également les communiquer aux 2 autres entités.

2.2.2 Partie Réseau

Pour faire communiquer ces 3 entités nous utilisons les *Sockets* pour la communication en réseau.
Par défault, dans le programme l'adresse ip du serveur *canal* est "127.0.0.1", c'est l'adresse dite *Localhost* car elle est interne à la machine et permet de manipuler du réseau dans sa propre machine.

2.2.3 Partie Grapique

Une partie graphique! et oui monsieur! Après nous avoir mis au défis de faire une interface graphique pour ce projet, nous en avons fait une avec SDL `https://www.libsdl.org`.
Elle execute l'émetteur, le canal et le recepteur avec le codage de hadamard graphiquement.



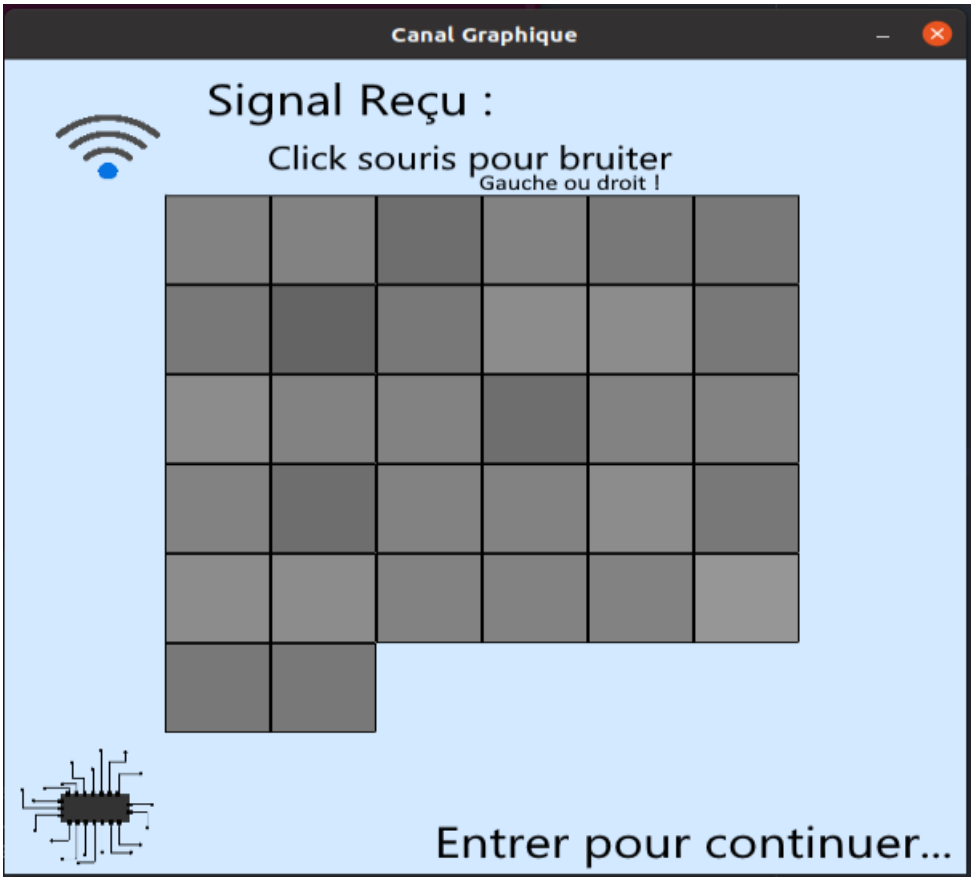
L'image précédente représente l'affichage de l'émetteur. Nous pouvons voir quelques personnages et une série de messages. En effet nous avons opté pour un affichage plutôt circulaire. Chaque personnage représente chaque utilisateur. Ils sont placé sur un cercle et chaqu'un est à une distance calculée en fonction du nombre d'utilisateur.

$\begin{cases} x = xo + r * cos(t) \\ y = yo + r * sin(t) \end{cases}$ Avec (xo,yo) sont les coordonnées du centre du cercle et t le rayon du cercle.

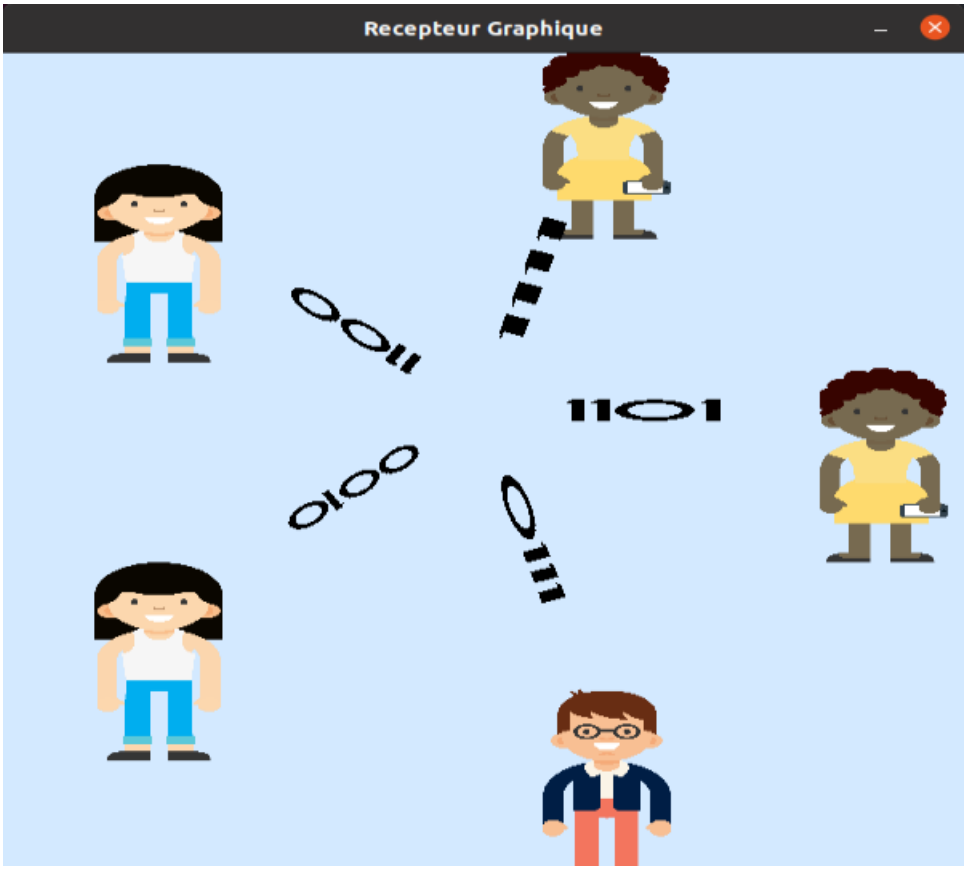
Nous avons, à l'aide de cette formule, pu calculer les coordonnées de chaque personnages.

La série de chiffre les reliants au centre sont leurs messages.

Par la suite quelques animations et une nouvelle fenêtre apparait pour laisser place à l'image suivante :



Celle-ci représente le signal reçu par canal. Les couleurs représentent une valeur de ce signal qui ne veut un dire mais l'objectif est de rendre la simulation plus visuelle.
La couleur est calculé comme si :
$valeur(signal) + 125\%255$.
Cette formule, un peu magique, fait pas grand chose en fait . Dans un premier temps, on "essaie" de passer le signal en positif, de lui ajouter une valeur 125 et dans un second temps, on lui applique un modulo (255).
Expliqué différement, on applique une transformation à la valeur du signal pour que si elle tend vers 0, elle aura un couleur vers du noir alors que si elle tend vers 255 elle sera plus blanche.
Pour finir le signal est envoyé au recepteur qui peut décoder le signal bruité.



On voit sur l'image ci-dessus tous les messages décodés par le recepteur. Si le bruit est trop fort/important le sigal sera certainement altéré.

2.3 Mise en pratique

Pour une meilleure compréhension de notre projet nous avons réalisé toutes les parties dans un fichier ".c" qui lui correspond [1].

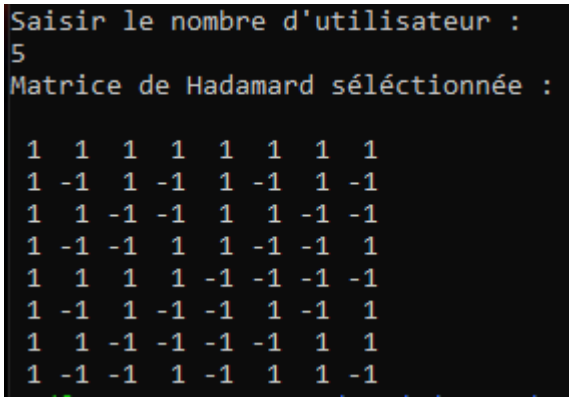
2.3.1 Fonctions de Hadamard

Les fonctions de Hadamard représentent la base de ce qui permettra de coder et de décoder les signaux. Nous avons conçu une seule fonction principale. C'est elle qui

1. Par exemple les fonctions de codage et de décodage sont dans "src/Encodage.c"

créer la matrice de Hadamard la plus petite possible en fonction du Nombre Utilisateur, il faut utiliser au minimum la mémoire. Pour générer la matrice, on l'initialise de taille 1 avec pour unique valeur 1.
Par la suite on repète sur la fonction la méthode suivante :
Tant que la matrice n'est pas suffisamment grande pour accueillir tous les utilisateurs, doubler la taille de la matrice pour chaque case : si elle n'est pas défini, on applique un calcul permettant de determiner sa valeur.

Cette mise en place permet de simplifier la construction de la matrice (un seul parcours, pas de "recopiage", donc compléxité en $O(n^2)$).
Il est aussi prévu d'autre plus petites fonctions pratiques : une pour détruire la matrice et une autre pour la visualiser (pour les tests).



Matrice de hadamard générée avec 5 utilisateurs

Un autre exemple est présent en annexe 4.5.

2.3.2 Fonctions d'encodage

Comme indiqué dans la partie précédente, les fonctions de générations de matrice de Hadamard sont très importantes pour le projet, en effet elles nous permettent de générer un message codé (non crypté) pour pouvoir ensuite l'envoyé sur le *Canal*.
Pour réaliser cette fonction d'encodage, on s'est servi du cours, rien de plus simple. On parcourt le message en entier ainsi que la ligne de Hadamard correspondante à l'utilisateur. Si le bit n du message est un 0 alors on inscrit l'inverse de la ligne de hadamard sinon on inscrit la ligne.
Ainsi avec ce procédé, on obtient un message codé de taille : *Taille_message* x la puissance de 2 supérieure (-1) au *nombre d'utilisateur*.

2.3.3 Fonctions de génération de code cycliques

Pour réaliser les fonctions de génération de code cycliques, nous avons suivi le cours et appliqué en C les algorithmes vues en cours.
Dans un premier temps, il nous fallait un vecteur (Tableau) contenant notre polynôme générateur.
Par la suite, nous avons défini notre Vecteur d'initialisation (première séquence) avec que des 1 comme ça l'algorithme fonctionne tout le temps.
Puis pour finir nous avons fait une fonction qui réalise toutes les séquences à partir de la séquence initiale. C'est-à-dire, qu'elle applique le polynôme générateur (2 puissance Taille_Séquence -1) fois.
Par exemple une séquence de taille 10, la liste finale sera de longueur 10 puissance 2 -1 c'est-à-dire 1023. Une copie d'un test de génération de code cyclique de taille 10 est présent en Annexe 4.1.

2.3.4 Codeur de Gold et JPL

Pour Gold et JPL, c'est la même chose nous appliquons les algorithmes vues en cours. Présent dans le fichiers *generateur.c*, ils se nomment CodeurJPL et CodeurGold.
Pour JPL, nous avons créer une fonction qui permet de tester si les tailles des matrices passées en paramètres sont bien première entre-elles.
Des exemples sont présent en annexes 4.2 et 4.3.

2.3.5 Fonctions de génération de nombre pseudo-aléatoires

Après avoir vu les fonctions de génération de code cycliques, nous avons créé quelques fonctions pour les rendres utiles.
En effet jusque là nos messages envoyés dans nos différents tests étaient générés aléatoirement par la fonction *rand()* présente dans la librairie *stdlib*. C'était très bien, très pratique mais bon il faut passer aux choses serieuses !
Nous avons voulu faire la même chose c'est-à-dire créer une fonction d'initialisation et une de génération.
Pour réaliser ces fonctions nous avons donc commencer par établir ce qu'il nous fallait

- Un code cyclique suffisament long.
- Une variable *static* qui permettrait de navigué dans ce cycle.
  Cela implique de l'initialiser de façon "non prévisible" pour éviter la répétition.

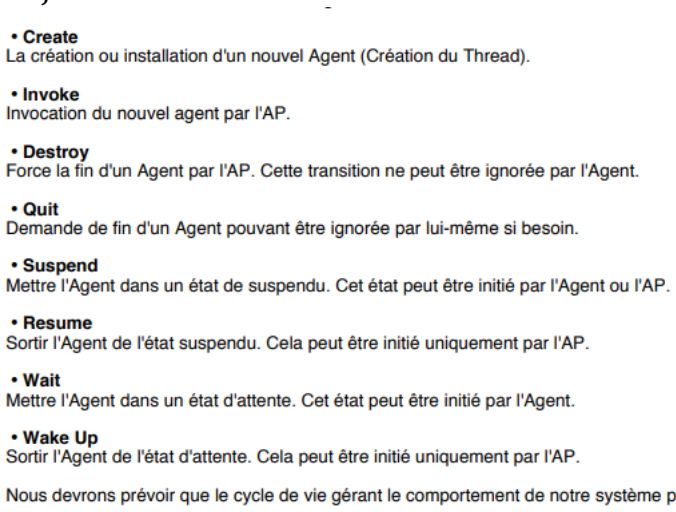
Disponible dans le fichier "generateur.c" les 3 fonctions Rand* sont décrites ci dessous pour mieux comprendre leurs particularitées.
RandInit() :
Génère un code cyclique [2] de taille 10 avec un Vecteur d'initialisation remplit de 1 et un polynôme générateur : I(x) = $1+x^3+x^4+x^5+x^6+x^7+x^8+x^9+x^{10}$. Cette séquence (code cyclique) générée est associée à un pointeur en variable globale nommé *seqRand*.
RandTerminer() :
Comme expliqué précédement la séquence générée est gardée par un pointeur en globale donc il faut impérativement libérer la mémoire pointée par ce pointeur à la fin du programme et cette fonction permet justement d'éviter d'appeler des *free()* [3].
RandGenerer() :
Pour finir, cette fonction est la plus importante de toutes. En effet c'est elle qui va nous générer notre nombre pseudo-aléatoire. Dans un premier temps, il faut créer une variable *static* qui est initialiée avec la variable time().
C'est une façon simple et comme cette fonction renvoie le temps depuis le 1er Janvier 1970, elle évolue constament et est donc quasiment imprévisible pour une personne qui ne s'en donne pas les moyens.
Pour génerer notre nombre, nous parcourons la ligne correspondante à l'indice [4] et nous ajoutons à notre valeur $(314159265*2^{compteur})$.
Pour être plus clair, prennons la séquence n°20 [5] (0100111000), on utilise un compteur qui parcours ce vecteur et si on rencontre (1) alors on applique le calcul précédent (Voir generateur.c).
De plus, nous utilisons une variable de type *long* codée sur 4 octets [6], c'est largement suffisant mais il peut arrivé dans certains cas que la valeur dépasse le maximum (2 147 483 647) alors la variable passera dans le négatif. Pour y remedier, nous avons mis une condition pour la valeur de retour qui sera mise positive si elle est négative. Ce calcul est totalement arbitraire et peut largement être amélioré.

2.3.6 Threads

Lors de la dernière séance de TP, nous avons tenté de manipuler des threads.
Via le site developpez.com [7], nous avons créer un petit programme permettant de

2. Un extrait du code cyclique est présent en Annexe 4.1.
3. Fonction pour libérer de l'espace mémoire alloué précédement
4. La valeur de la variable *static*
5. Vois Annexe 4.1
6. Paquet de 8 bits
7. https ://franckh.developpez.com/tutoriels/posix/pthreads/

faire des threads.
Dans le dossier *Thread* du projet, il y a differents fichiers permettant de réaliser ce que le sujet demandait à savoir :



- **Create**
La création ou installation d'un nouvel Agent (Création du Thread).

- **Invoke**
Invocation du nouvel agent par l'AP.

- **Destroy**
Force la fin d'un Agent par l'AP. Cette transition ne peut être ignorée par l'Agent.

- **Quit**
Demande de fin d'un Agent pouvant être ignorée par lui-même si besoin.

- **Suspend**
Mettre l'Agent dans un état de suspendu. Cet état peut être initié par l'Agent ou l'AP.

- **Resume**
Sortir l'Agent de l'état suspendu. Cela peut être initié uniquement par l'AP.

- **Wait**
Mettre l'Agent dans un état d'attente. Cet état peut être initié par l'Agent.

- **Wake Up**
Sortir l'Agent de l'état d'attente. Cela peut être initié uniquement par l'AP.

Nous devrons prévoir que le cycle de vie gérant le comportement de notre système peut être évolutif.

Pour que le programme ait toujours un accès sur ces thread pour leur communiquer des informations, nous avons opté pour une liste.
Bien que cela nous semblait avancer, nous n'avons pas pu terminer ce tp car il nous manquais du temps.
Nous n'avons pas pu utiliser toutes les fonctionalitées disponibles avec la librairie *pthread.h* mais il aurait été évident d'avoir recours à un mutex pour la réalisation du tp.
Sinon pour la réalisation nous avons remarqué que la fonction de création d'un thread *pthread_create (pthread_t * thread, pthread_attr_t * attr, void * (* start_routine) (void *), void * arg)*; prend en paramètre une fonction *(* start_routine)* et également un paramètres à savoir *arg*. Avec cela nous savions comment envoyer un message au thread n. Cette façon est certe correct mais dans notre cas il nous a semblé plus simple de passer par la liste c'est-à-dire d'avoir une variable dans un élément de la liste qui est rataché au thread et qui contiendrait la variable de communication [8].
Par la suite, il nous suffit simplement de passer en paramètre de la création le numero du thread, il pourra alors consulter la liste pour avoir accès à ses informations.

8. Variable utilisée pour faire communiqué des informations entre le main et le thread

Note

Les parties suivantes n'ont pas de rapport direct avec le sujet de TP mais comme nous nous en sommes servis, il nous a semblé bon de faire un petit point dessus.

2.3.7 Réseau

Pour faire communiquer nos différents programmes nous avons opté pour l'utilisation des *Sockets* en C qui sont assez simple d'utilisation.
Voir Annexe 4.1 pour plus détails. Comme l'objectif du TP n'est pas d'utiliser le réseau, nous avons créer des fonctions plus simples.
Elles sont détaillés dans l'archive du TP dans le fichier *réseau.c*.
Pour illustrer prennont un exemple.
On peut voir en Annexe 4.4 que une socket à besoin d'être crée pour fonction (*socket()*) puis a besoin dêtre connectée (*connect()*). Bien, pour utiliser ces fonctions nous avons créer des alternatives :

- *SocketCreation() -> socket()*.
- *SocketConnexion() -> connect()*.
- *SocketCreationConnexion(int) ; socket() + connect()*.

Le paramètre (int) de la fonction *SocketCreationConnexion(int)* est utilisé pour préscisé s'il sagit d'un client ou d'un serveur auquel cas la fonction fera soit un *connect()* (Client) ou un *bind()* (Serveur).

On voit encore une fois qu'il faut impérativement démarer en premier le serveur (Canal) avant les clients (Emetteur et Recepteur).

2.3.8 SDL

Sans trop entrer dans les details, nous avons utilisé pour la partie graphique, la SDL.
La SDL est une bibliothèque graphique libre de droits disponible en C, C++ ...
Facile d'utilisation, nous avons utilisé certaines de ses nombreuses fonctions.
Pour ne pas tout mélanger, nous avons créer un sous-répertoire nommé "Graphique" (cf 2.1 Organisation des fichiers). Dans ce répertoire, nous avons re-créer les fichiers d'executable principaux à savoir **canal.c**, **emetteur.c** et **recepteur.c**. Ces derniers appellent les modules déjà compilés plus haut c'est-à-dire ceux sans le "programme principal", nous n'allions pas refaire les fonctions déjà créées. A celà, nous avons rajouté un nouveau fichier nommé **graphics.c** qui contient toutes les fonctions principales que nous nous sommes servis pour la partie graphique.

Sans trop entrer dans les détails, elles sont au nombre de 7 :

- *drawMessage* : Elle permet de générer aléatoirement un personnage et le placer sur la fenêtre (cf 2.2.3).
- *reduire* : Animation graphique avec l'image de paquet qui fait reduire, donnant l'impression d'une compression.
- *grandir* : Fonction qui fait l'inverse de la fonction reduire.
- *envoyer* : Cette fonction permet de faire déplacer l'image centrale de paquet (déjà réduit) vers la gauche pour envoyer vers le prochain programme.
- *recevoir* : Fonciton qui fait l'inverse de la fonction envoyer ; Déplacement de la gauche vers le millieu.
- *changerCouleur* : Cette fonction permet de changer la couleur en fonction de la valeur du signal, elle est utilisée dans l'affichage de l'ajout de bruit dans le signal.
- *bruiter* : La fonction affiche, sur la fenêtre, un rectangle par valeur du signal reçu par le canal. Cette fonction a surtout pour but de faire participer l'utilisateur à l'ajout de bruit dans le signal.

Toutes ces fonctions sont bien évidement disponible dans l'archive du projet dans Graphics/graphics.c.

2.4 Fichiers de tests

Il y a beaucoup de fonctions à faire et comme beaucoup dépendent d'autre, nous avons tout de suite pensé à faire des petits fichiers de tests permettant de pouvoir tester chaque module interne au projet indépendamment.
Il sont au nombre de 7.
Sur ce tableau une courte explication de ce que chaque fichier permet de tester.

"src/test_*"	Breve description
Encodage	Tester les fonctions de codage d'un ou plusieurs message(s), le décodage. Le fichier permet également de tester la somme de tous les messages.
Generateur	Permet de tester les différents codeurs réalisés pendants les séances : *JPL - Gold - HDBn*.
Hadamard	Fonctions de Hadamard qui génère la matrice de Hadamard en fonction du **nombre d'utilisateur**.
Random	Generation de nombre pseudo-aléatoires.
HDBnAri	2 en 1 : Les fonctions HDBn et Arithmétiques.
Réseau_Client	Tester l'envoie d'un message en réseau.
Réseau_Serveur	Tester la reception d'un message en réseau.

3 Conclusion

4 Annexes

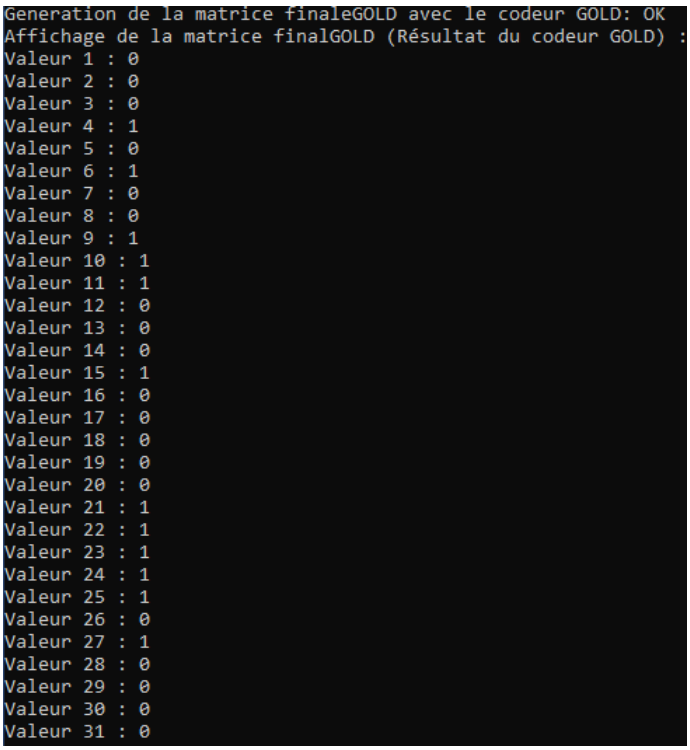
4.1 Extrait de génération de code cyclique de taille 10

```
Valeur 10 : 0 0 1 1 0 1 0 0 0 1
Valeur 11 : 0 0 0 1 1 0 1 0 0 0
Valeur 12 : 1 0 0 0 1 1 0 1 0 0
Valeur 13 : 1 1 0 0 0 1 1 0 1 0
Valeur 14 : 1 1 1 0 0 0 1 1 0 1
Valeur 15 : 0 1 1 1 0 0 0 1 1 0
Valeur 16 : 0 0 1 1 1 0 0 0 1 1
Valeur 17 : 1 0 0 1 1 1 0 0 0 1
Valeur 18 : 0 1 0 0 1 1 1 0 0 0
Valeur 19 : 1 0 1 0 0 1 1 1 0 0
Valeur 20 : 0 1 0 1 0 0 1 1 1 0
Valeur 21 : 0 0 1 0 1 0 0 1 1 1
Valeur 22 : 1 0 0 1 0 1 0 0 1 1
Valeur 23 : 0 1 0 0 1 0 1 0 0 1
Valeur 24 : 1 0 1 0 0 1 0 1 0 0
Valeur 25 : 1 1 0 1 0 0 1 0 1 0
Valeur 26 : 1 1 1 0 1 0 0 1 0 1
Valeur 27 : 0 1 1 1 0 1 0 0 1 0
Valeur 28 : 0 0 1 1 1 0 1 0 0 1
Valeur 29 : 1 0 0 1 1 1 0 1 0 0
Valeur 30 : 0 1 0 0 1 1 1 0 1 0
Valeur 31 : 0 0 1 0 0 1 1 1 0 1
Valeur 32 : 1 0 0 1 0 0 1 1 1 0
Valeur 33 : 0 1 0 0 1 0 0 1 1 1
Valeur 34 : 0 0 1 0 0 1 0 0 1 1
Valeur 35 : 0 0 0 1 0 0 1 0 0 1
Valeur 36 : 1 0 0 0 1 0 0 1 0 0
Valeur 37 : 0 1 0 0 0 1 0 0 1 0
Valeur 38 : 0 0 1 0 0 0 1 0 0 1
Valeur 39 : 1 0 0 1 0 0 0 1 0 0
Valeur 40 : 0 1 0 0 1 0 0 0 1 0
Valeur 41 : 0 0 1 0 0 1 0 0 0 1
Valeur 42 : 1 0 0 1 0 0 1 0 0 0
Valeur 43 : 0 1 0 0 1 0 0 1 0 0
Valeur 44 : 0 0 1 0 0 1 0 0 1 0
Valeur 45 : 1 0 0 1 0 0 1 0 0 1
Valeur 46 : 1 1 0 0 1 0 0 1 0 0
Valeur 47 : 0 1 1 0 0 1 0 0 1 0
Valeur 48 : 1 0 1 1 0 0 1 0 0 1
Valeur 49 : 0 1 0 1 1 0 0 1 0 0
```

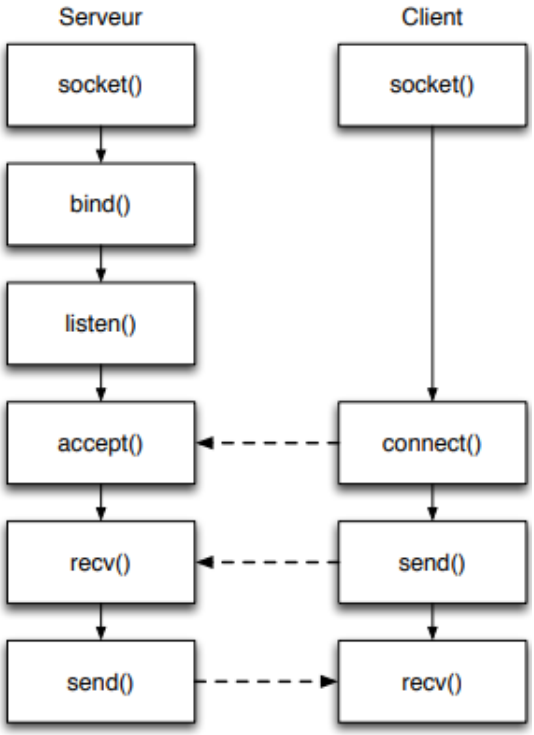
4.2 Exemple d'utilisation de la fonction CodeurJPL

```
Generation de la matrice finale avec le codeur JPL: OK
Affichage de la matrice final (Résultat du codeur JPL) :
Valeur 1 : 1 1 1 1 1 1 1 1 1 1
Valeur 2 : 0 1 0 1 1 0 1 1 1 1
Valeur 3 : 1 0 1 0 1 0 0 1 1 1
Valeur 4 : 1 1 0 1 0 1 0 0 1 1
Valeur 5 : 0 1 0 0 1 1 1 0 0 1
Valeur 6 : 1 0 1 0 0 0 1 1 0 0
Valeur 7 : 1 1 1 1 0 1 0 1 1 0
Valeur 8 : 0 1 1 1 1 0 1 0 1 1
Valeur 9 : 1 0 0 1 1 0 0 1 0 1
Valeur 10 : 1 1 1 0 1 1 0 0 1 0
Valeur 11 : 0 1 0 1 0 0 1 0 0 1
Valeur 12 : 1 0 0 0 1 0 0 1 0 0
Valeur 13 : 1 1 1 0 0 0 0 0 1 0
Valeur 14 : 0 1 1 1 0 0 0 0 0 1
Valeur 15 : 1 0 1 1 1 1 0 0 0 0
Valeur 16 : 1 1 0 1 1 0 1 0 0 0
Valeur 17 : 0 1 1 0 1 1 0 1 0 0
Valeur 18 : 1 0 0 1 0 0 1 0 1 0
Valeur 19 : 1 1 0 0 1 1 0 1 0 1
Valeur 20 : 0 1 1 0 0 1 1 0 1 0
Valeur 21 : 1 0 1 1 0 1 1 1 0 1
Valeur 22 : 1 1 1 1 1 0 1 1 1 0
Valeur 23 : 0 1 0 1 1 1 0 1 1 1
Valeur 24 : 1 0 1 0 1 1 1 0 1 1
Valeur 25 : 1 1 0 1 0 0 1 1 0 1
Valeur 26 : 0 1 0 0 1 0 0 1 1 0
Valeur 27 : 1 0 1 0 0 0 0 0 1 1
Valeur 28 : 1 1 1 1 0 1 0 0 0 1
Valeur 29 : 0 1 1 1 1 1 1 0 0 0
Valeur 30 : 1 0 0 1 1 1 1 1 0 0
Valeur 31 : 1 1 1 0 1 1 1 1 1 0
Valeur 32 : 0 1 0 1 0 1 1 1 1 1
Valeur 33 : 1 0 0 0 1 0 1 1 1 1
Valeur 34 : 1 1 1 0 0 0 0 1 1 1
Valeur 35 : 0 1 1 1 0 1 0 0 1 1
Valeur 36 : 1 0 1 1 1 1 1 0 0 1
Valeur 37 : 1 1 0 1 1 0 1 1 0 0
Valeur 38 : 0 1 1 0 1 1 0 1 1 0
Valeur 39 : 1 0 0 1 0 0 1 0 1 1
Valeur 40 : 1 1 0 0 1 0 0 1 0 1
Valeur 41 : 0 1 1 0 0 1 0 0 1 0
Valeur 42 : 1 0 1 1 0 0 1 0 0 1
Valeur 43 : 1 1 1 1 1 0 0 1 0 0
Valeur 44 : 0 1 0 1 1 0 0 0 1 0
Valeur 45 : 1 0 1 0 1 0 0 0 0 1
Valeur 46 : 1 1 0 1 0 1 0 0 0 0
Valeur 47 : 0 1 0 0 1 0 1 0 0 0
Valeur 48 : 1 0 1 0 0 1 0 1 0 0
Valeur 49 : 1 1 1 1 0 0 1 0 1 0
```

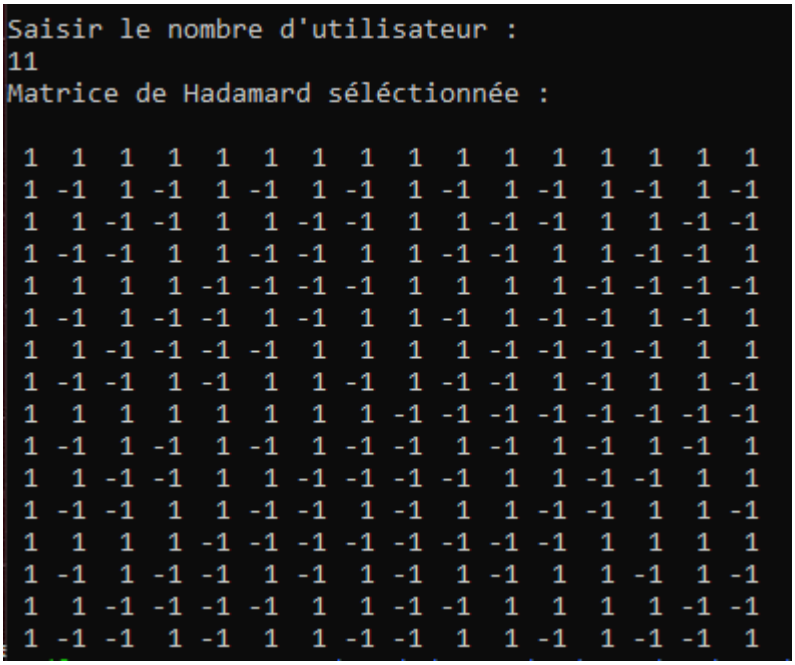
4.3 Exemple d'utilisation de la fonction CodeurGold



4.4 Représentation shématique des sockets en C



4.5 Matrice de hadamard générée avec 11 utilisateurs



5 Liens

Threads : https://franckh.developpez.com/tutoriels/posix/pthreads/
SDL : https://www.libsdl.org
Sockets : https://broux.developpez.com/articles/c/sockets/