



Projet de jeu en réseau

Rapport final

Arthur ZUCKER, Younes BELKADA

MAIN3A - 2019

Polytech Sorbonne
MAIN3A
2018-2019



Nous tenons à remercier notre encadrant, le Professeur François Pêcheux, pour ses précieux conseils ainsi que pour sa bonne humeur lors des nombreux cours que nous avons eus avec lui cette année universitaire.

Date de rendu, 31 Mai 2019

Table des matières

Introduction	5
1 La théorie du "threading"	7
1.1 Un peu d'architecture des ordinateurs	7
1.1.1 La notion de processus	8
1.1.2 OS : Kernel, Userland et les appels système	9
1.2 Le Fork	10
1.3 Les Threads (non ça ne se mange pas)	10
1.3.1 Principe	11
1.3.2 Pthread, les threadsdans le langage C	11
2 La théorie des réseaux	13
2.1 Définition et organisation du réseau internet	13
2.2 L'adressage, les protocoles et la communication	13
2.2.1 Le modèle de communication en couches (TCP/IP)	14
2.2.2 Le principe d'encapsulation	14
2.2.3 Le protocole HTTP	16
2.3 Le réseau en C : les sockets	16
3 Cahier des charges du projet	17
3.1 SH13 ou SherlockHolmes 13	17
3.1.1 Les règles du jeu	17
3.1.2 Les règles de programmation	19
3.2 Programmes fournis	20
3.2.1 SendMessageToClient / SendMessageToServer	20
3.2.2 Autres fonctions relatives à la création d'un jeu	21
4 Notre réalisation	23
4.1 Fonctionnement du code : threading et Réseau	23
4.2 Le code que nous avons écrits	23
4.3 Ressources complémentaires	24

5	Améliorations	27
5.1	Mise à jour automatique du serveur	27
5.1.1	Rajout dans le protocole	27
5.1.2	Quels sont les traitements à réaliser ?	27
5.2	Chat global	28
5.2.1	Contexte	28
5.2.2	Comment cela se traduit-il sur le protocole ?	29
5.2.3	Sur le code ?	29
5.3	Chat - message privé	31
5.3.1	Contexte	31
5.3.2	Comment se traduit-il au niveau du code ?	31
6	Conclusion	33
	Bibliographie	35

Introduction

Contexte

Dans le cadre de notre première année de MAIN, nous avons tous deux été initiés, au cours des trois derniers mois, à la programmation réseau en C. Dans le cadre du cours encadré par M. Pêcheux, nous avons participé à un projet de re-création du protocole **HTTP** en se basant sur un jeu existant, le **Sherlock Holmes 13**. Il faut savoir que ce jeu est basé entièrement sur la programmation réseau en C, notion que nous aborderons dans ce rapport.

A quoi sert la programmation réseau en C ?

La programmation réseau est un peu le fantasme de (presque) tous les programmeurs... On a tous déjà rêvé de faire tourner notre propre programme permettant de communiquer à travers un réseau, que ce soit sur internet ou en local.



FIGURE 2 – *Counter Strike* : un jeu auquel nous avons tous déjà joué qui se base typiquement sur le concept de réseau

Il faut admettre que la programmation réseau (surtout en C !) est assez complexe : elle nécessite la compréhension de la théorie des réseaux, de la théorie des processus et du fonctionnement d'un ordinateur. Cependant, une fois ces notions maîtrisées les applications sont très diverses et très pratiques (exemple : création de chat) ouvrant plusieurs portes dans le monde de la programmation.

Pré-requis

Nous avons écrit ce rapport dans une optique de présentation de résultats, mais aussi d'introduction à différents concepts informatiques. C'est pour cela qu'aucune connaissances en réseau ou fonctionnement

d'un ordinateur n'est nécessaire à la compréhension de notre rapport. Ainsi si le lecteur se trouvait dans une situation d'incompréhension, nous le prendrions comme un échec de notre part.

Afin de pouvoir étudier le fonctionnement d'un serveur, l'introduction au fonctionnement d'un ordinateur et à la théorie des réseaux est nécessaire.

Nous introduirons aussi la notion de parallélisme (et donc de *threading*) qui est primordiale pour la gestion d'un serveur, puis nous expliquerons ce que sont les protocoles, comment fonctionne le réseau internet, de l'écriture d'un message sur une application (ou dans un terminal dans notre cas) à la réception du message par un tiers, en passant par une description du protocole **HTTP** et du protocole "**Y&A**". Une fois cette partie théorique entièrement abordée, nous nous pencherons sur le réel sujet qui nous a été proposé et le cahier des charges que nous avons élaboré. Quelques schémas de fonctionnement seront proposés, ainsi que des captures d'écran de l'interface graphique de notre projet. Enfin nous présenterons certaines améliorations que nous avons eu le temps d'implémenter.



1. La théorie du "threading"

Le threading (ou filtage en français) est une technique informatique de programmation multi-tâches utilisée de nos jours dans la plupart des applications réseau. Cependant, afin de comprendre le fonctionnement des threads, il nous paraît important de rappeler quelques notions de programmation, ainsi que de présenter les outils utilisés avant l'apparition du threading.

1.1 Un peu d'architecture des ordinateurs

Le fonctionnement et l'architecture d'un ordinateur peuvent être décrits de manière assez concise. Il faut tout d'abord savoir que, d'un point de vue physique il existe trois composants fondamentaux :

- Le **processeur** s'occupe d'effectuer toutes les opérations arithmétiques et logiques, de répartir les temps d'utilisation entre chaque processus (à travers l'ordonnanceur) ou application, gère les mouvements de données et enfin les différents appels systèmes. Pour effectuer ces tâches il possède différents **registres** qui peuvent stocker des résultats : c'est la mémoire la plus rapide d'un processeur. Il ne possède qu'une quantité limitée de registres, qui détermine sa vitesse, mesurée en hertz. La mémoire des registres est dite **mémoire vive statique**, car elle utilise un type de stockage appelé la SRAM (Static Random Access Memory). Cependant, le processeur a très souvent besoin de communiquer avec la mémoire de stockage pour récupérer des informations stockées de manière intemporelle. Or la vitesse d'accès à la mémoire est très souvent inférieure à celle du processeur. C'est là qu'entre en jeu la mémoire cache.
- L'**antémémoire** (ou simplement mémoire cache) copie temporairement des données issues de la mémoire et étant fréquemment consultées, pour la représenter "en gros". Le cache est créé pour aller aussi vite que le processeur, ainsi il permet un intermédiaire entre le processeur et la mémoire de stockage, permettant de ne pas perdre de vitesse. Dans 98% des cas où le processeur a besoin d'instructions, le cache parvient à les copier. Ce taux descend à 94% pour les données.
- La **mémoire** quant à elle peut être de différent type, mais de nos jours on parle souvent de mémoire flash qui est un type de **mémoire de stockage** de masse sur lequel on peut réécrire.

Le schéma suivant (Figure 1.1) présente la communication entre ces trois composants, selon le schéma de Von Newman¹.

1. cf. Cours M. Pecheux

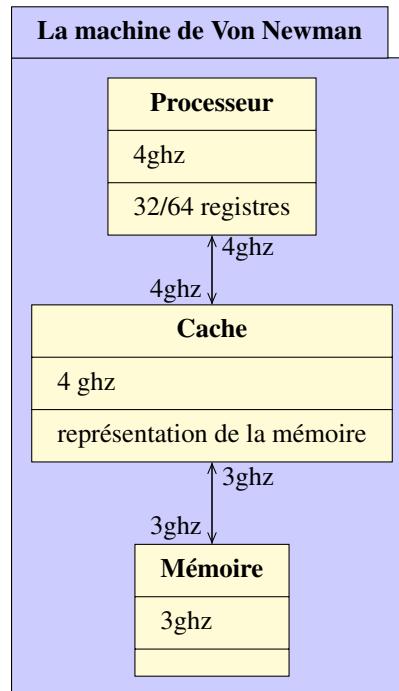


FIGURE 1.1 – Représentation de la communication entre le processeur la mémoire et le cache.

Le fonctionnement simplifié d'un ordinateur étant désormais introduit, nous pouvons aborder la notion de processus.

1.1.1 La notion de processus

Un processus est un programme, un ensemble d'instructions souvent chargé dans la mémoire vive d'un ordinateur depuis la mémoire de stockage, et prêt à être exécuté. Plusieurs processus sont souvent exécutés *en même temps* par un processeur. Or en réalité, ceux-ci ne sont jamais exécutés réellement en même temps (du moins pour les ordinateurs n'ayant qu'un seul processeur) pourtant, lorsque l'on utilise un ordinateur, on a l'impression que tout se fait en même temps : ma page firefox sur laquelle j'écris ce rapport est active, mais mon application de musique est aussi active, et elle ne s'arrête pas ! Ce qui nous donne cette impression de parallélisme c'est l'**ordonnanceur** (scheduler en anglais), et le principe de **pseudo-parallélisme**.

Afin de rendre l'explication plus claire, nous l'illustrerons par des schémas qui seront visibles dans les pages suivantes. **L'ordonnanceur** est un programme contenu dans le cœur du système d'exploitation d'un ordinateur, permettant de gérer le temps d'exécution accordé à chaque processus. Il permet de mettre en place le pseudo-parallélisme en donnant à chaque processus un temps d'exécution très court, mais très souvent. En tout temps, il élit le processus suivant qui devrait s'exécuter afin d'assurer le bon fonctionnement de l'interface avec l'utilisateur. Le cycle de vie d'un processus est donc le suivant :

1.1. Un peu d'architecture des ordinateurs

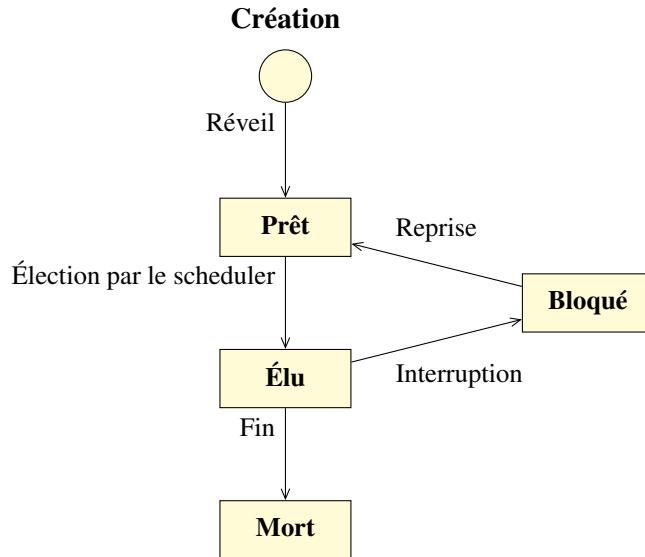


FIGURE 1.2 – Diagramme d'état d'un processus

Or même s'il permet ce choix, un processus qui est bloqué doit pouvoir reprendre son exécution sans encombre. Son contexte d'exécution, c'est à dire les états des registres lors de l'arrêt du processus, doit être stocké ! Enfin le système d'exploitation doit être en mesure de récupérer et de restaurer le contexte du processus élu. Le contexte de chaque processus est stocké dans le tableau des processus, qui se trouve lui aussi dans le noyau (ou kernel) abordé dans le paragraphe suivant.

1.1.2 OS : Kernel, Userland et les appels système

Le système d'exploitation (Operating system en anglais) d'un ordinateur est un ensemble de programmes qui dirigent l'utilisation "physique" des composants d'un ordinateur. Il fait la liaison logiciel/matériel. Afin de protéger l'ensemble de la mémoire de l'ordinateur, d'accélérer la gestion des processus et de gérer l'espace de stockage autorisé pour chaque processus, l'OS sépare la mémoire en deux zones : l'**Userland**, visible par tous les processus, et le **Kernel** ou noyau, invisible pour les autres processus, qui s'occupent respectivement de la gestion des appels systèmes pour l'un, et de la gestion des processus, des fichiers, des utilisateurs et des périphériques pour l'autre. Comme expliqué précédemment, les contextes d'exécution des processus sont stockés dans le noyau, ainsi ils ne sont pas accessibles depuis n'importe quelle application : lorsque je suis en train de taper des informations sur mon journal intime, mes entrées qui seront stockées dans la mémoire ne sont pas accessibles par Skype, par exemple, qui s'exécute aussi.

Avec toutes ces informations, on peut désormais représenter un schéma du pseudo-parallélisme, en prenant par exemple deux applications : l'horloge et firefox, retrouvé sur la figure 1.3

Les appels systèmes sont effectués par les applications qui cherchent à obtenir de l'aide auprès du système d'exploitation ou un accès à des informations stockées et gérées par le noyau. L'ensemble des ces appels, qui peuvent être de différentes natures sont proposés par le noyau du système d'exploitation : ils sont "programmés" dans le noyau. Ces appels sont très souvent codés en C de manière assez intuitive, et ils sont toujours gérés par le noyau de l'OS.

Or parmi ces appels système il y en a deux qui nous intéressent : le fork et le thread, qui sont des appels liés à la gestion de processus.

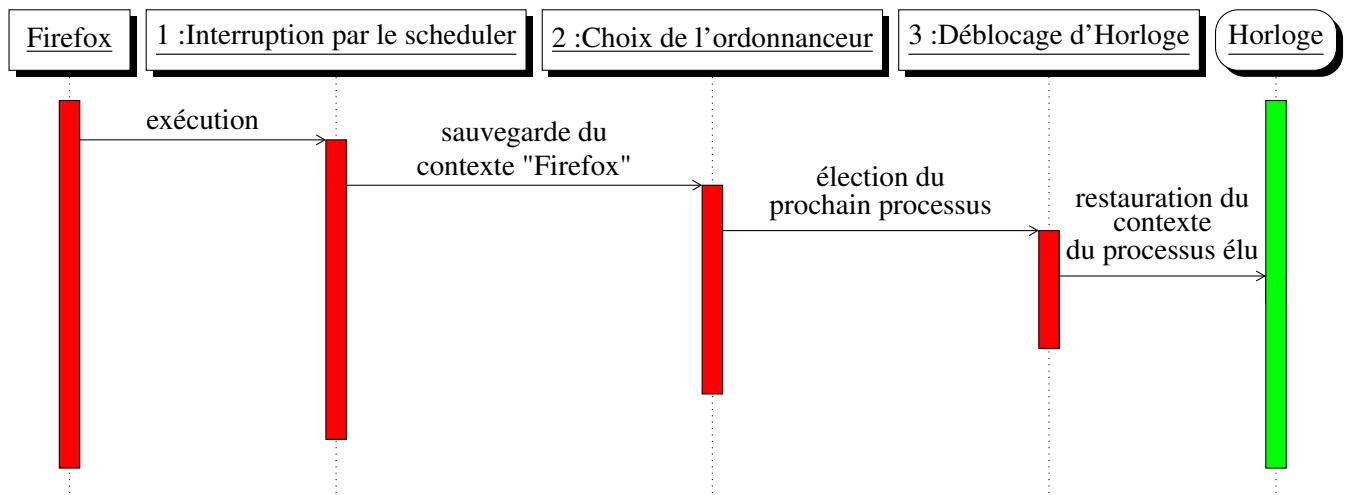


FIGURE 1.3 – Diagramme d'exécution de processus

1.2 Le Fork

Le Fork est un appel système permettant à un processus (appelé processus père) de donner naissance à un nouveau processus (processus fils) qui est sa copie conforme. Ainsi le processus fils est identique au processus père, seul sont PID (Process IDentifier) et sont PPID (Parent Process IDentifier) est différent. On effectue souvent un Fork pour paralléliser un programme demandant beaucoup de temps de calcul. Ainsi même si l'ordonnanceur stop le processus père (il l'arrête pour conserver le pseudo parallélisme) le processus fils peut être élu, et il peut ainsi effectuer des calculs à la place du père.

Lors de l'appel-système `fork()`, le noyau effectue les opérations suivantes

- Il crée une nouvelle ligne dans la table des processus
- Il copie le contexte d'exécution du processus père sur cette nouvelle ligne
- Il alloue un PID au processus fils
- En général il lance aussi l'exécution de ce processus fils
- Il renvoie au processus ayant donné naissance le PID de son fils

Comme vu précédemment, le but du fork est généralement d'obtenir plus de parallélisme, et les processus fils créés ont besoin de communiquer avec leur père (que ferait-on sans les précieux conseils de nos anciens...). De ce fait, pour que deux processus communiquent ensemble ils doivent utiliser les IPC (inter-process communication en anglais). Il peuvent être de différents types, mais dans le cadre d'un Fork, le père attend souvent que son fils ait fini sa tâche avant de continuer à s'exécuter par exemple. Ceci se traduit par `pid_t wait` qui indique que le père attend que son fils finisse son exécution. Si un processus fils est terminé, mais que le père n'a pas attendu la fin de son exécution (souvent une fois qu'un fils a fini de travailler on le tue), on assiste à la création d'un processus zombie ! Le processus fils continue de s'exécuter, alors qu'il n'a plus rien à faire !

Le Fork est très puissant mais possède aussi de nombreux inconvénients : chaque processus d'un fork possède son propre contexte d'exécution, qu'il est coûteux de dupliquer, c'est pourquoi on le remplace souvent par des thread (fils) qui sont décrits dans la section suivante.

1.3 Les Threads (non ça ne se mange pas)

Même si une explication détaillée peut être très compréhensible, rien ne vaut un bon schéma. C'est pourquoi nous présenterons un modèle de fonctionnement de thread après avoir abordé l'aspect pratique de cet appel système.

1.3. Les Threads (non ça ne se mange pas)

1.3.1 Principale

Les threads (ou fil d'exécution) sont similaires aux processus : ils regroupent des instructions à exécuter. Cependant, les threads sont appelés processus légers de par la rapidité de leur création (souvent 100× plus rapide que pour la création d'un fork). Un thread est créé à partir d'un processus, par un processus et partage son contexte d'exécution avec le processus. Il a donc accès à toutes variables du processus principal. Puisqu'il n'y a pas de changement au niveau de la mémoire virtuelle lors de l'exécution d'un thread, le passage d'un contexte d'exécution à un autre (context switch) entre deux threads est très rapide. Cet ensemble de caractéristiques rend le thread très pratique pour la manipulation de serveurs : lorsqu'un serveur reçoit une demande de communication, il devait tout de suite la résoudre il ne pourrait accepter une autre communication. Or nous détestons quand un site est inaccessible ! les threads permettent de déléguer certaines tâches.

Les threads peuvent être créés avant même que leur utilisation soit requise ! On parle de **threadpool** pour piscine à thread. La création des thread pourrait ralentir le fonctionnement d'un serveur, c'est pourquoi avant même d'avoir reçu une demande de connexion, un serveur possède déjà tous les outils nécessaires pour répondre aux attentes du client.

Comme expliqué ci-dessus, toutes les variables des threads sont partagées : c'est la mémoire partagée. Même si elle permet au programme principale d'aller beaucoup plus vite, cela pose des problèmes. Quand deux threads cherchent à modifier deux variables en même temps, ou quand l'accès à une variable est demandé par deux threads, on est face à un sérieux problème. Or le mécanisme de synchronisation avec le **mutex** permet de résoudre ce problème !

Le mutex permet de verrouiller l'accès aux variables, et oblige donc certains threads à attendre le déverrouillage du mutex, pour accéder à une variable. Tout cela paraît très abstrait, mais dans la partie suivante nous verrons une petite implémentation en C qui rendra le tout compréhensible.

1.3.2 Pthread, les threads dans le langage C

Dans le langage de programmation C, la gestion de threads se fait à l'aide de la librairie *Pthread*. Un thread est représenté par une variable de type `p_thread` et est créé à l'aide de la commande suivante :

```
int pthread_create(pthread_t * thread, NULL, void *(*start_routine) (void *), void *arg);
```

Si la création du thread fonctionne, cette fonction retourne 0, n'importe quelle valeur sinon. La signification des paramètres est la suivante : le premier paramètre est un pointeur vers un thread, le deuxième paramètre ne nous intéresse pas, et le troisième et quatrième paramètre sont un pointeur vers la fonction (donc les instructions que devra exécuter le thread) et les arguments pris en entrée par cette fonction. Pour éviter d'avoir des processus zombie, il est habituel d'utiliser une fonction permettant d'attendre la fin d'un thread appelé le `pthread_join()`. Elle permet aussi de récupérer la valeur de retour du thread, indiquant si la terminaison du thread était normale ou forcée (cela permet de prévenir le mauvais fonctionnement du programme).

Nous avons maintenant les outils nécessaires pour créer un programme utilisant des thread ! Mais nous allons nous attaquer au problème de synchronisation. Les **mutex** sont aussi gérés par la librairie `pthread`. Nous les utilisons dans notre programme ! Lorsque l'on connaît la zone qui sera accédée par plusieurs thread, on parle de zone critique. On la traite en verrouillant le mutex lors de l'utilisation par un thread, qui s'empresse toujours de déverrouiller le mutex une fois son travail effectué.

La fonction `int pthread_mutex_lock (pthread_mutex_t * mutex);` permet de déterminer le début d'une zone critique, la fin de la zone étant déterminée par la fonction `int pthread_mutex_unlock (pthread_mutex_t * mutex);`. L'accès à la mémoire est maintenant protégé, il ne nous reste plus qu'à synchroniser les threads en permettant à certains thread de s'endormir, puis d'être réveillés uniquement lorsqu'une certaine condition est respectée. On parle de *conditions*, elles sont représentées par les `struct pthread_cond_t` en C. Grossièrement elle permettent à un thread d'attendre le signal envoyé par un autre thread avant de continuer son exécution.

Pour résumer toutes ces notions, le diagramme de séquence suivant Figure 1.4 exprime comment fonctionne un programme ayant plusieurs thread synchronisés. La syntaxe exacte du code C n'est pas respectée,

mais cela permet une compréhension intuitive. Ici le main souhaite calculer un gros nombre tout en l'écrivant sur un fichier. Il possède un thread qui effectuera les calculs, et un thread qui écrira le résultat du calcul dès que le thread rendra la main sur le résultat.

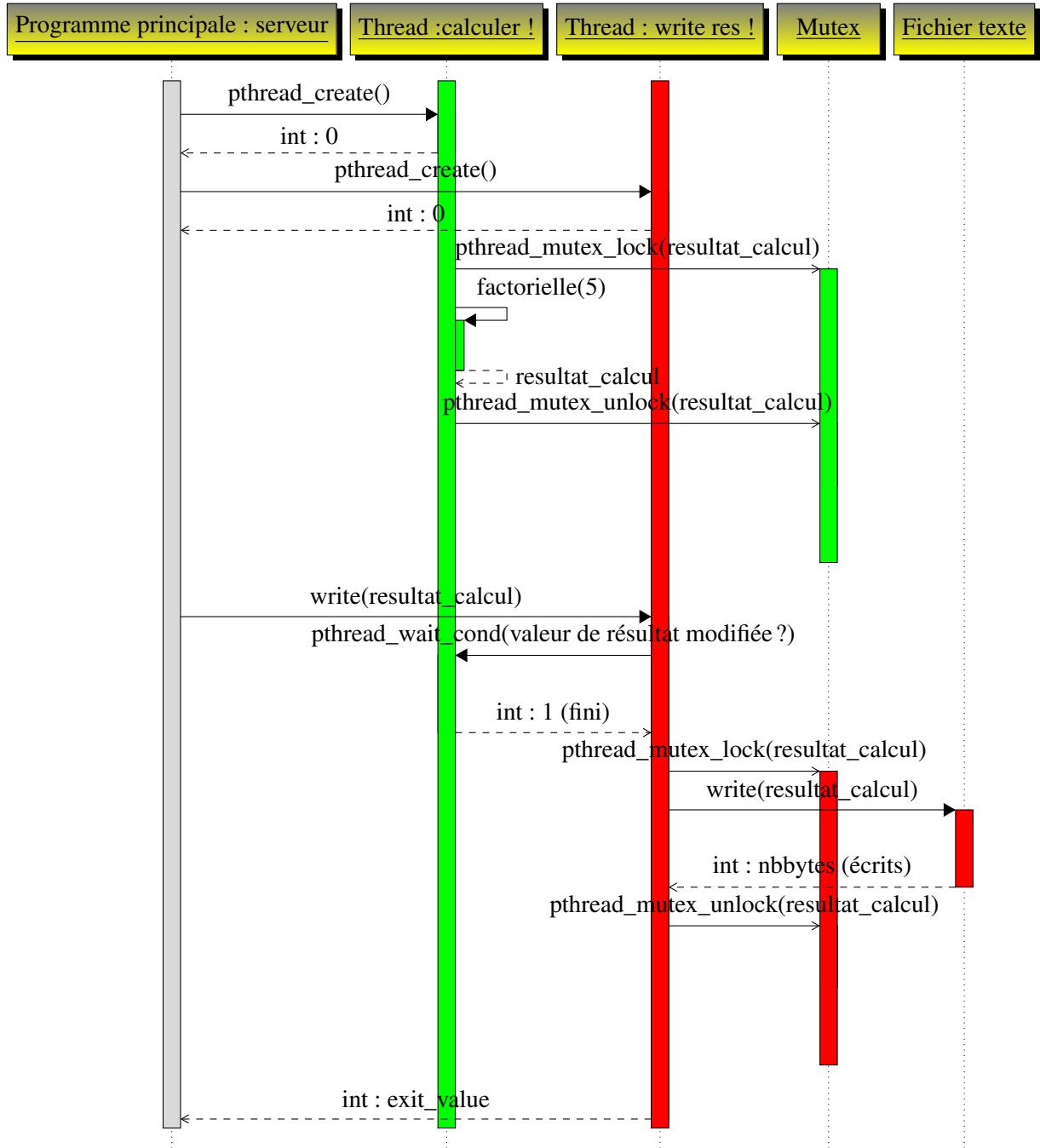


FIGURE 1.4 – Fonctionnement d'un petit programme de calcul



2. La théorie des réseaux

Il est de nos jours quasiment impossible de vivre sans cet outil magique qu'est Le réseau : le monde entier est connecté grâce à lui. Cependant peu de personnes sont capables d'expliquer le fonctionnement physique mais aussi logistique qui se cache derrière. Nous tenterons de donner les éléments importants au fonctionnement des réseaux, pour ensuite étudier les méthodes existantes sous C permettant d'utiliser un réseau.

2.1 Définition et organisation du réseau internet

Le réseau internet est avant tout un réseau informatique (public et accessible mondialement), c'est un ensemble de machines inter-connectées entre elles pouvant s'échanger des informations. Il existe bien évidemment différents types de réseaux, qui sont souvent liés à leur portée respective. Ce qui différencie les réseaux peut aussi bien se situer dans leur mode de fonctionnement que dans les composantes utilisées pour transmettre l'information. Le réseau internet qui désigne le plus grand réseau au monde est caractérisé par un regroupement de très nombreux sous-réseaux !

Pour qu'une machine puisse se connecter à un réseau, elle a besoin d'un petit objet électronique appelé la carte réseau. Lorsqu'un client souhaite recevoir des données, celle-ci seront reçues à l'aide de cette carte. La question que tout le monde se pose est : comment un serveur sait-il à qui envoyer quoi et comment ? On peut aussi se demander comment le message est envoyé d'un point de vue logiciel.

2.2 L'adressage, les protocoles et la communication

Pour qu'une machine puisse communiquer sur un réseau, elle doit disposer d'une adresse IP (Internet Protocole), qui est une suite de nombres codés en général sur 4 bits. Par exemple 192.159.234.21 est une adresse IP. Chaque machine en possède une (de manière un peu simpliste, nous ne décrirons pas le sub-netting ici) qui l'identifie pour le réseau internet. Afin d'envoyer et de recevoir des données, il faut aussi que chaque machine parle le même langage, et l'ensemble des règles qui définissent ce langage sont les protocoles. Ils sont utilisés dans les différentes couches d'abstraction : celles-ci correspondent en fait à des logiciels intermédiaires, que l'on ne voit pas. Nous ne voyons que la couche applicative, par exemple l'application mail, mais ne sommes jamais en contact direct avec les sous couches qui permettent de traiter le texte, de le traduire en langage binaire, de l'envoyer etc

Il existe de nombreux protocoles, et leurs usages peuvent aller de l'envoi d'un mail à la protection des informations transmises. Nous nous intéresserons ici aux protocoles de communication les plus utilisés pour transmettre des données sur internet : les suites de protocole TCP/IP. Dans ceux-ci est inclus le protocole HTTP.

2.2.1 Le modèle de communication en couches (TCP/IP)

La communication entre deux machines suit des règles strictes. L'ensemble des protocoles utilisés classiquement lors de la communication entre deux machines est regroupé sous le nom de TCP/IP. Il est important de retenir qu'un modèle de communication à été développé afin de faciliter la compréhension des utilisateurs du réseau, il s'agit de la représentation d'un ordinateur en couches d'abstraction, communiquant entre elles. La figure (2.2) présente ce modèle en détails.

Les protocoles utilisent ce que l'on appelle les numéros de port, qui correspondent souvent à des programmes particuliers : lorsque que l'on envoie ou reçoit une information grâce à internet, le logiciel doit être capable de dire pour quelle application est l'information. Par exemple lorsque j'écris un mail, les données envoyées vers mon interlocuteur contiendront le port correspondant aux mails. Dans la figure suivante (2.2), nous montrerons comment est créé le message envoyé.

2.2.2 Le principe d'encapsulation

Lors de son passage entre chaque couche d'abstraction une information supplémentaire est ajoutée à la donnée initiale. Et lorsque le message passe dans l'autre sens, dans les couches respectives de l'autre machine, ces informations sont analysées et supprimées du message. Ces informations supplémentaires sont appelées des headers et permettent de transmettre des informations sur le protocole qui a été utilisé par chaque couche.

On appelle cela le principe d'encapsulation¹.

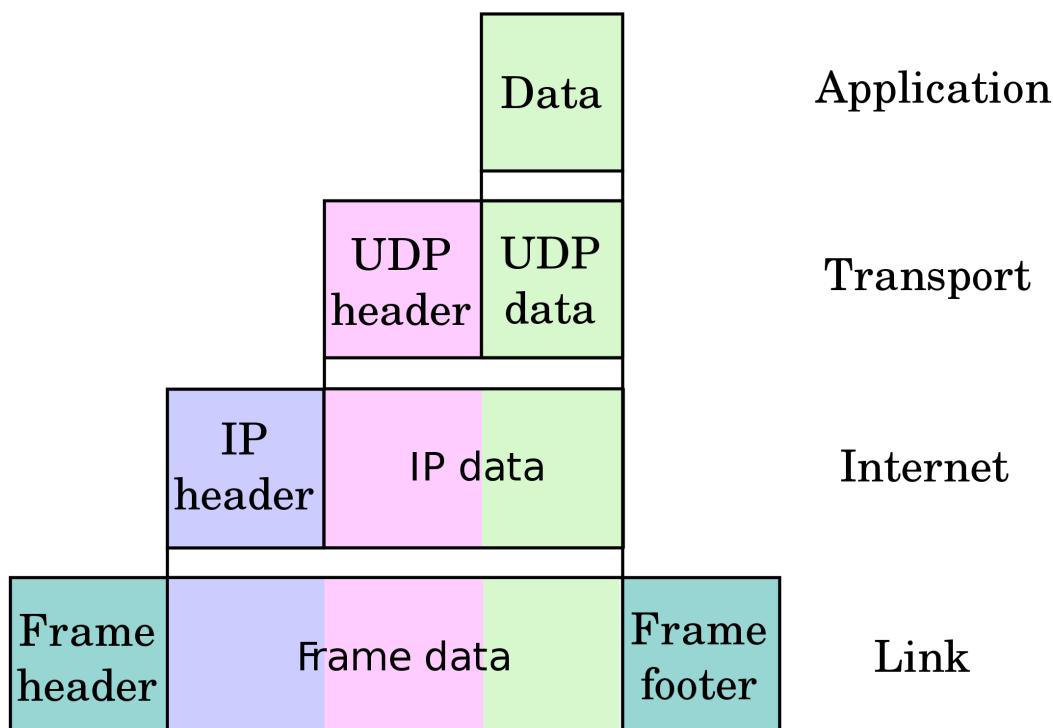


FIGURE 2.1 – L'encapsulation des données, exemple avec le protocole UDP¹

Pour notre application, seuls cinq informations sont nécessairement présentes dans un message :

- Le port que la source utilise
- Le port que la cible utilise ou port de destination

1. [https://en.wikipedia.org/wiki/Encapsulation_\(networking\)](https://en.wikipedia.org/wiki/Encapsulation_(networking))

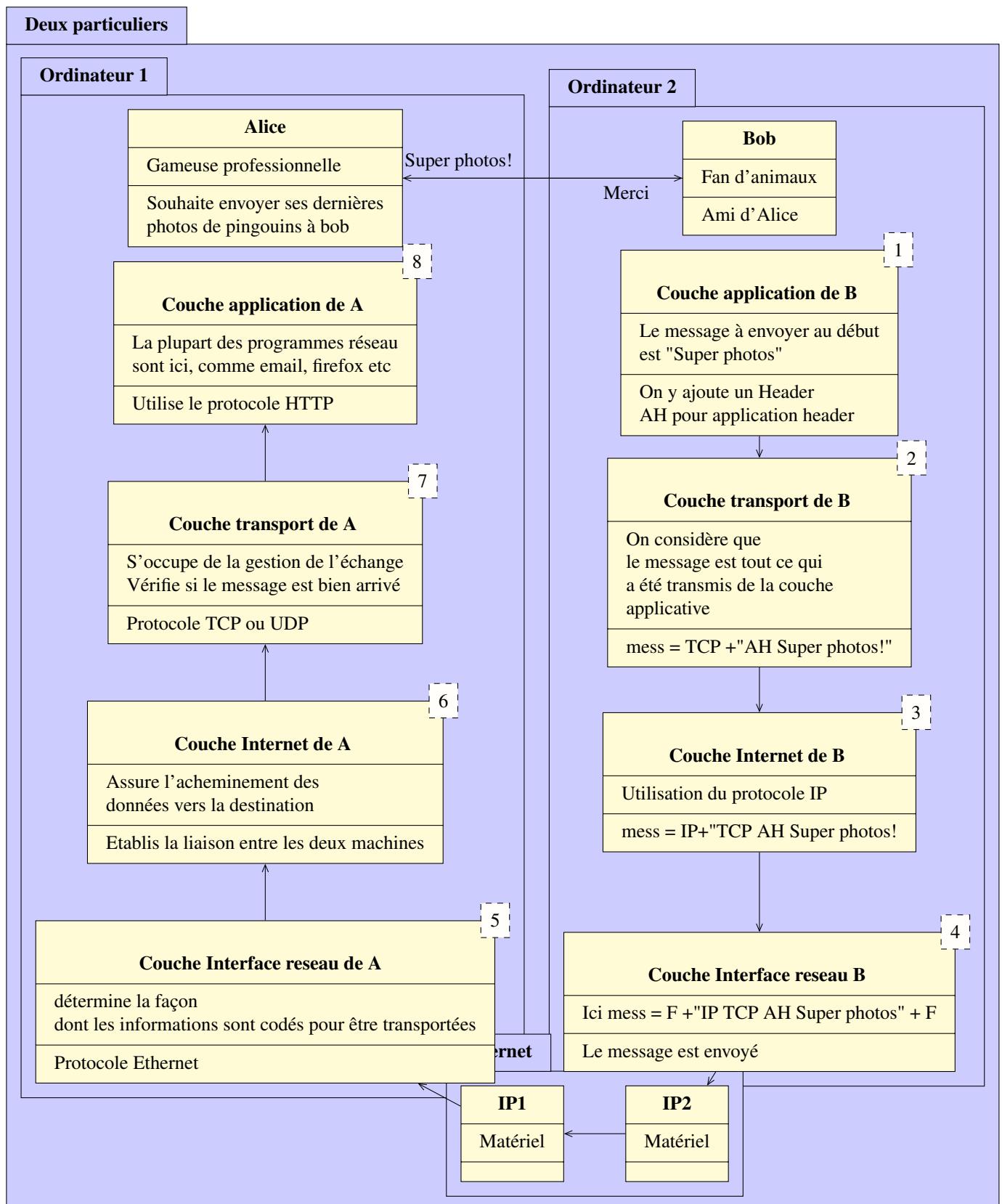


FIGURE 2.2 – Modèle TCP/IP

- L'adresse IP de la cible, c'est à dire l'adresse sur le réseau internet du destinataire
- L'adresse IP de la source, pour que le destinataire sache de qui vient l'information
- Le message !

2.2.3 Le protocole HTTP

Le protocole **HTTP** (HyperText Transfer Protocol en anglais) utilisé dans la couche applicative est implémenté pour le World Wide Web (internet). Ce protocole définit un ensemble de règles et de méthodes utilisables par le client et les réponses correspondantes envoyées par le serveur. Le but de notre projet est de créer ces règles et donc en quelque sorte de re-créer le protocole **HTTP**. Pour cela, nous devons donc utiliser le langage C afin de créer une relation entre 2 machines. Le but est de définir plusieurs règles et plusieurs méthodes permettant une bonne communication entre deux machines !

2.3 Le réseau en C : les sockets

Sous Unix, "Tout est fichier"², même les connexions internet. Celles-ci peuvent se traduire comme des flux de données, et donc comme des fichiers. Les sockets sous Linux remplissent ce rôle : une socket est une structure abstraite permettant la communication entre deux processus ou applications à travers un serveur. La structure d'une socket est la suivante :

```
struct sockaddr_in {
    uint8_t         sin_len;      /* longueur totale      */
    sa_family_t     sin_family;   /* famille : AF_INET    */
    in_port_t       sin_port;    /* le numéro de port    */
    struct in_addr  sin_addr;    /* l'adresse internet   */
    unsigned char   sin_zero[8]; /* un champ de 8 zéros */
};
```

Listing 1 – structure d'une socket

Nous verrons en détail comment nous avons utilisés cette structure dans la partie 4.

2. François Pêcheux

3. Cahier des charges du projet

Cette partie présente le cahier des charges du projet que nous avons élaboré.

3.1 SH13 ou SherlockHolmes 13

3.1.1 Les règles du jeu

Le jeu que nous allons présenter ici se nomme *Sherlock Holmes 13* il s'agit d'un jeu de société multijoueurs qui se joue à 4 à travers le réseau. Il faut savoir qu'une version "papier" existe et coûte aux alentours de 15 euros sur le marché.



FIGURE 3.1 – La version papier du jeu SH13¹

Le miracle de la programmation réseau nous permet de jouer à ce jeu à distance via internet, ou sur un réseau local (moins amusant...). Afin de pouvoir rejoindre une partie, il est nécessaire de connaître l'adresse IP du serveur ainsi que le port utilisé par ce dernier pour jouer. Le jeu débute une fois que 4 joueurs se sont connectés au serveur. Chaque joueur reçoit ensuite trois cartes comme celle présentée sur la figure 3.2 ci-dessous

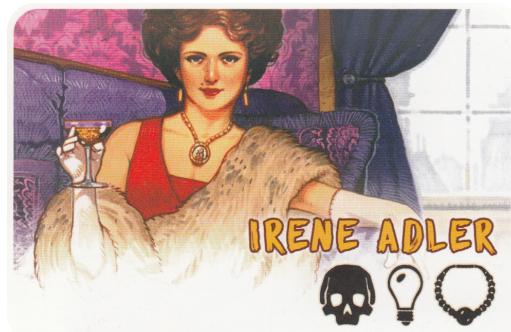


FIGURE 3.2 – Un exemple de carte qu'un joueur peut recevoir

Comme nous pouvons le voir, chaque carte contient des attributs, sur celle-ci : un collier, un crâne et une lampe. Le jeu contient 13 cartes, et chaque joueur reçoit 3 cartes, il y a donc une carte qui n'est pas distribuée et qui correspond au **coupable** que les joueurs doivent trouver. À chaque tour les joueurs ont trois possibilités d'action :

- Demander à l'ensemble des joueurs s'ils possèdent ou non des cartes avec un certain attribut. Cependant le joueur n'aura pas accès au nombre d'attributs que possèdent les joueurs (sinon ça serait trop facile).
- Demander à un joueur en particulier la quantité exacte d'un attribut (par exemple le joueur 1 demande au joueur 2 combien de lampes il possède).
- Déclarer directement qui est le coupable. Dans le cas où le joueur a raison, il gagne directement la partie.

1. source : <https://boardgamegeek.com/boardgame/149869/sherlock-13>

3.1. SH13 ou SherlockHolmes 13

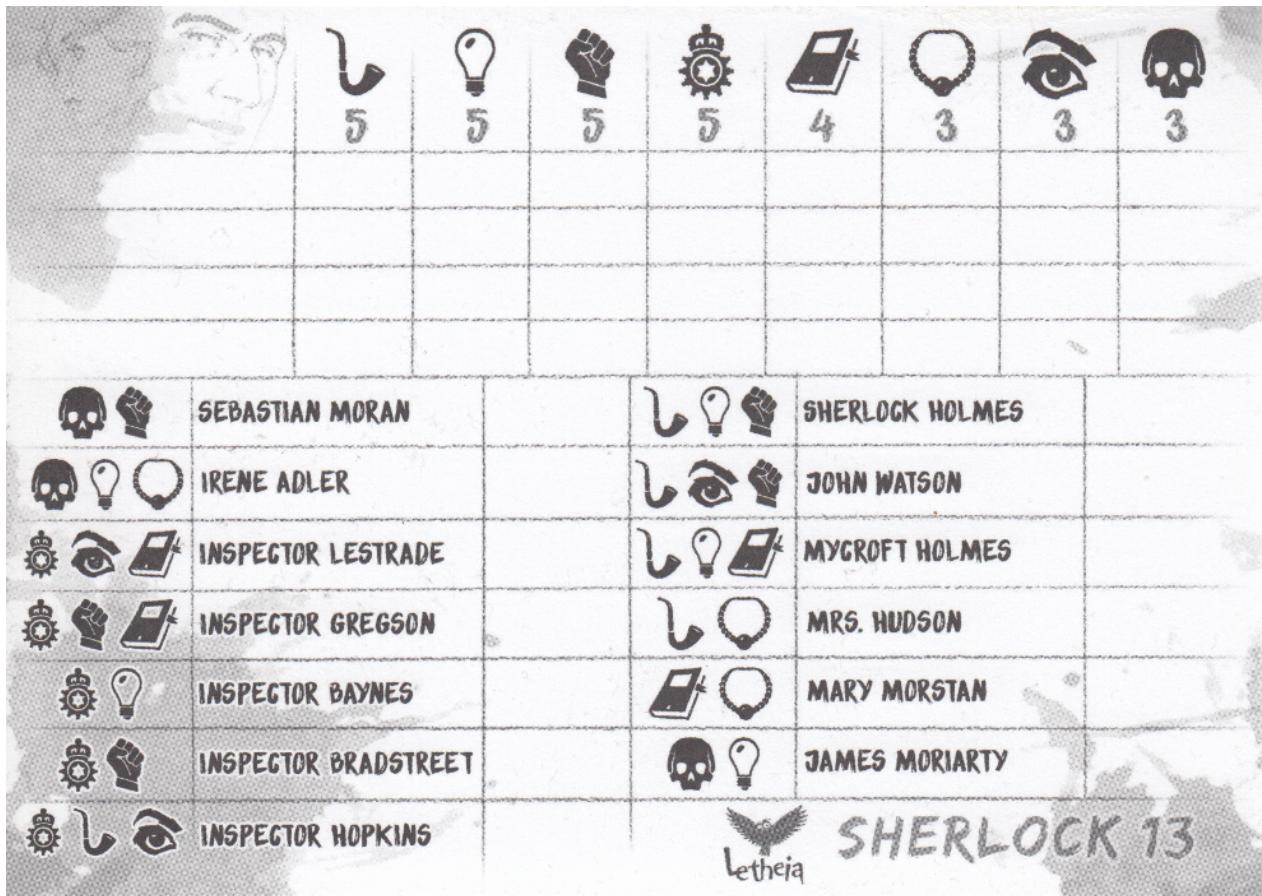


FIGURE 3.3 – Grille de jeu de SH13

Comme nous pouvons le voir sur la grille ci dessus, pour chaque attribut on a accès à la quantité de ce dernier dans le "deck". Au fur à mesure que le jeu avance, le tableau de jeu se remplit et il sera donc possible, à partir d'un certain temps de déroulement du jeu, de déduire le nom du coupable et de gagner !

3.1.2 Les règles de programmation

Tout se fait exclusivement via le terminal pour pouvoir lancer le jeu. Dans le doute, effectuer les commandes suivantes avant de commencer :

```
1 Test-SH13 $> make clean
2 Test-SH13 $> make
```

Comme annoncé plus haut, il faut d'abord lancer un serveur qui va pouvoir accueillir les quatre joueurs. On lance donc la commande (en choisissant comme port le port 32000)

```
1 Test-SH13 $> ./server 32000
```

Une fois le server créé, les joueurs doivent connaître l'adresse IP ainsi que le port du server pour pouvoir se connecter, (la commande ipconfig sur le terminal permet de connaître son adresse IP). Nous allons supposer ici que les quatre joueurs sont sur le même ordinateur (ce qui est très peu probable .. !) afin de nous faciliter l'écriture des commandes sur le terminal. Supposons que le joueur Bill souhaite rejoindre la partie. Il devra

cependant connaître le port qu'il souhaite utiliser pour la connexion. Pour des soucis de pratique, on estimera que chaque joueur se connecte sur le port 32001, 32002 etc.

Revenons à Bill. Il devra donc taper la commande suivante :

```
1 Test-SH13 $> ./sh13 localhost 32000 localhost 32001 Bill
```

Le *localhost* permet de spécifier que l'adresse IP est au fait l'adresse locale de l'ordinateur courant. Le *localhost* qui suit le port 32000 permet de spécifier au programme que Bill est au fait sur l'ordinateur d'adresse IP locale, donc sur l'ordinateur courant. De même, si le joueur Boule souhaite rejoindre cette merveilleuse partie, il devrait taper la commande :

```
1 Test-SH13 $> ./sh13 localhost 32000 localhost 32002 Boule
```

Boule ne pourra pas choisir le port 32001 puisqu'il est déjà utilisé par Bill. Dans le cas où il tente de faire cela, Boule recevra un *bind error* dû aux sockets : le port étant déjà pris, la socket ne sera pas valide.

3.2 Programmes fournis

Entrons maintenant dans le vif du sujet, le jeu est constitué de deux fichiers principaux, *sh13.c* et *serverc.c*. Le premier consiste au programme qui sera exécuté par les clients, et le second par le serveur. Ces fichiers étaient initialement remplis de fonctions indispensables à la programmation réseau, nous allons expliquer l'utilité de quelques unes (les plus essentielles), le reste des fonctions est détaillé dans la documentation créée à l'aide de **doxygen** (voir le chapitre 4 "Notre réalisation").

3.2.1 SendMessageToClient / SendMessageToServer

Afin de pouvoir communiquer via notre protocole, il est indispensable d'envoyer des messages aux clients et au server afin de "décrypter" ces derniers. Les fonctions citées ci dessus constituent donc le nerf du programme.

3.2. Programmes fournis

```

void sendMessageToServer(char *ipAddress, int portno, char *mess)
{
    int sockfd, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char sendbuffer[256];

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    server = gethostbyname(ipAddress);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
          (char *)&serv_addr.sin_addr.s_addr,
          server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
    {
        printf("ERROR connecting\n");
        exit(1);
    }

    sprintf(sendbuffer, "%s\n", mess);
    n = write(sockfd,sendbuffer,strlen(sendbuffer));

    close(sockfd);
}

```

Listing 2 – SendMessageToServer

La fonction prend en argument l'adresse IP du serveur sous forme de chaîne de caractères, le numéro de port du serveur ainsi que du message que nous souhaitons envoyer. Et enfin nous appliquons un write qui permet d'envoyer au socket le message entré en argument suivi d'un saut à la ligne. Afin de recevoir le message le serveur doit appliquer la fonction read.

La fonction SendMessageToClient reprend le même principe que ce programme (d'ailleurs le corps de la fonction est quasiment identique), et la fonction BroadcastMessage parcourt la liste des clients (tcpClients) et applique la fonction SendMessageToClients à chaque client.

3.2.2 Autres fonctions relatives à la création d'un jeu

Ces fonctions se situent uniquement sur le serveur, en effet c'est le serveur qui initialise le jeu (comme expliqué lors des premières parties, le serveur constitue le cerveau de la partie).

server.c/melangerDeck

Ce que l'on peut imaginer lorsqu'on lit le nom de cette fonction c'est que cette dernière mélange "littéralement" le deck, c'est à dire qu'il mélange le tableau contenant le nom de chaque carte (variable nommée *nom cartes*). Cependant le mélange du deck est plus subtile que cela...



Quoi de mieux qu'un deck bien mélangé pour une partie équitable...

Une variable nommée *deck* est présente dans le programme, cette dernière correspond en réalité aux indices des cartes ; pour chaque indice on associe une carte dans la variable *nom cartes* et ce qu'on mélange est au final l'ensemble des indices. Cette opération revient exactement au même que de mélanger un tableau, puisque pour chaque indice *i* du deck mélangé, il suffit de récupérer la carte d'indice *i*. La fonction *printDeck* permet d'afficher le deck une fois mélangé sur le terminal.

server.c/printClients et findClientByName

Comme l'ensemble des clients sont stockés dans la variable globale *tcpClients* la manipulation et la gestion de ces derniers devient relativement simple. Ces fonctions sont d'ordre pratique, afin de pouvoir les utiliser dans la fonction principale. Elles permettent respectivement d'afficher l'ensemble des clients connectés sur le serveur et de retourner l'id du joueur grâce à son nom.

Fonctions d'affichage SDL

L'interface graphique est entièrement gérée grâce à la librairie SDL, (en particulier SDL2.0) permettant de générer des surfaces et des interfaces graphiques. Cette librairie est largement utilisée pour la création de jeux par exemple, et cette dernière est très bien documentée sur le net.



4. Notre réalisation

Nous présenterons l'ensemble des fonctions que nous avons implémentés, ainsi que le fonctionnement complet du programme.

4.1 Fonctionnement du code : threading et Réseau

Coté serveur²

Le serveur ne possède pas de thread, mais il possède plusieurs fonctions qui lui permettent de communiquer avec les clients. Par exemple la fonction `sendMessageToClient` qui, à l'aide d'une socket établie une connexion avec un client, et lui envoie un message contenant le header (C,Z,Q,W,...) ainsi qu'un message. Une fois le message envoyé, la connexion est fermée. Il est simplement important de noter qu'il est précisé à la socket le numéro de port du client à utiliser, l'adresse IP du client et le type d'adressage utilisé. Enfin il faut lier la socket à un port pour la rendre valide.

Le serveur crée aussi dans son main, une première socket, qui représente la liste des demandes de connexion, et attend des demandes de la part des clients à travers le `listen(5)` qui indique que le serveur traite 5 demande au maximum. La socket est passive, puisqu'elle ne sert qu'à accepter des connexions. Lorsque le serveur accepte une nouvelle connexion (depuis la socket principale), il crée une nouvelle socket représentant cette connexion, sans avoir d'impact sur la socket principale. Ceci se fait avec le (`accept()`), ensuite le serveur traite ce qui a été envoyé : on voit apparaître le message "Received packet from..." .

Coté client

Le client possède un thread executant la fonction `<fn_serveur_tcp>`. Ce thread permet la communication entre le serveur et le client. Les clients sont toujours en train d'attendre un message du serveur, et ce thread s'occupe de la réception de ce message. Ce thread ne se finit jamais, et tant que les clients sont synchronisés, la socket d'écoute (passive) n'accepte pas de nouvelle socket. La variable volatile `syncro` gère le temps de traitement des informations : comme expliqué précédemment, les contexte d'exécution des thread sont partagés, `syncro` à besoin de passer à travers le cache pour bien être actualisé lors du passage d'un processus à un thread ou d'un thread à un autre. Dans le cas où un client reçoit un message du serveur, il actualise la valeur de cette variable à 1. Tant que `syncro` vaut 1, le client ayant reçu un message bloque `syncro` à l'aide d'un mutex, et analyse le message reçu par le serveur. Il change ensuite `syncro` à 0 lorsqu'il a fini. Cela permet de ne pas lire lorsqu'un client ne reçoit rien. De plus, cela laisse suffisamment de temps au client pour traiter un message, avant d'accepter une nouvelle information du serveur. Ainsi, tant que tous les clients n'ont pas traités la réponse du serveur, ils n'acceptent pas de nouvelles informations : personne ne peut être en avance par rapport aux autres.

4.2 Le code que nous avons écrits

Dans le code qui nous a été fournis, nous avons implémenté l'ensemble du protocole permettant les demandes liées au jeu, la gestion du joueur courant, l'affichage d'un chat, la gestion d'un abandon de joueur, la gestion de l'affichage des boutons "go", "chat" et "quit".

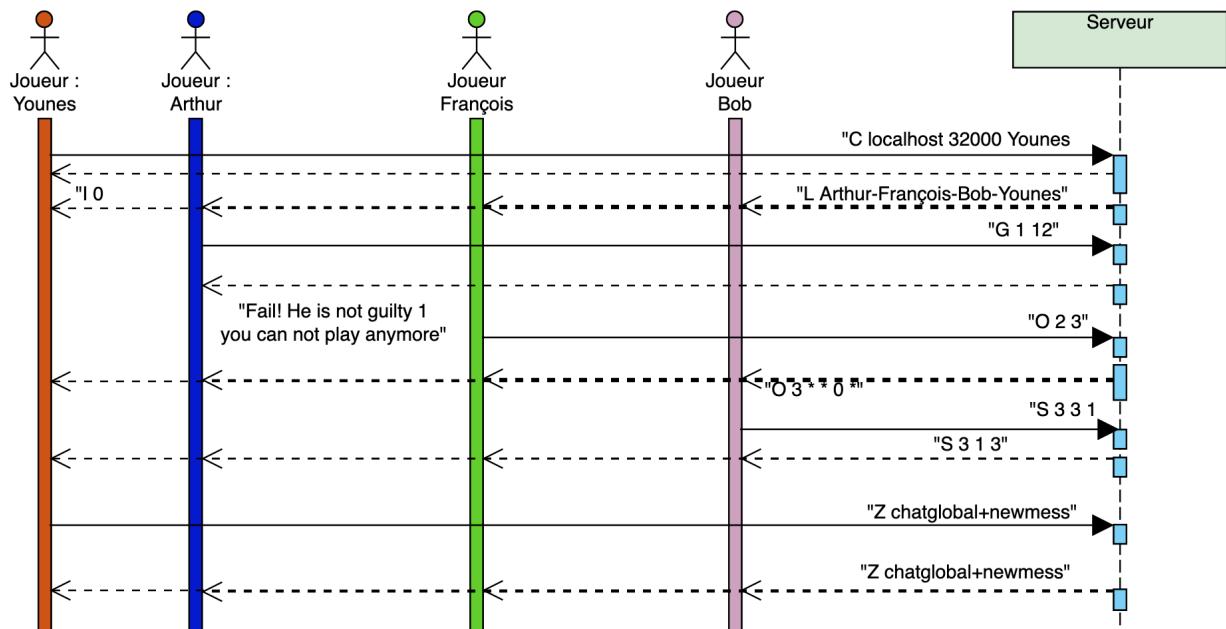


FIGURE 4.1 – Diagramme séquentiel du protocole implémenté

Diagramme de séquence du protocole que nous avons implémentés cas avec 2 joueurs Ici nous considérons 4 joueurs ayant lancés l’application sh13 et un serveur. Sont représenté toutes les interactions, ainsi toutes les information envoyé par les clients vers le serveur, et les retour du serveur. Cet ensemble de règle défini le protocole Y&A

4.3 Ressources complémentaires

Par curiosité, nous nous sommes renseigné sur la création d’une documentation automatique, et nous sommes familiarisés avec l’outil doxygen, permettant de traiter des commentaires spéciaux dans un code C, les utilisant pour aider à la manipulation et distribution du projet. Le lien vers le fichier html obtenu est disponible sur notre git : <https://github.com/younesbelkada/SH13> dans la partie Sources/html. Un aperçu de la page web obtenue est proposé avec la figure suivante.

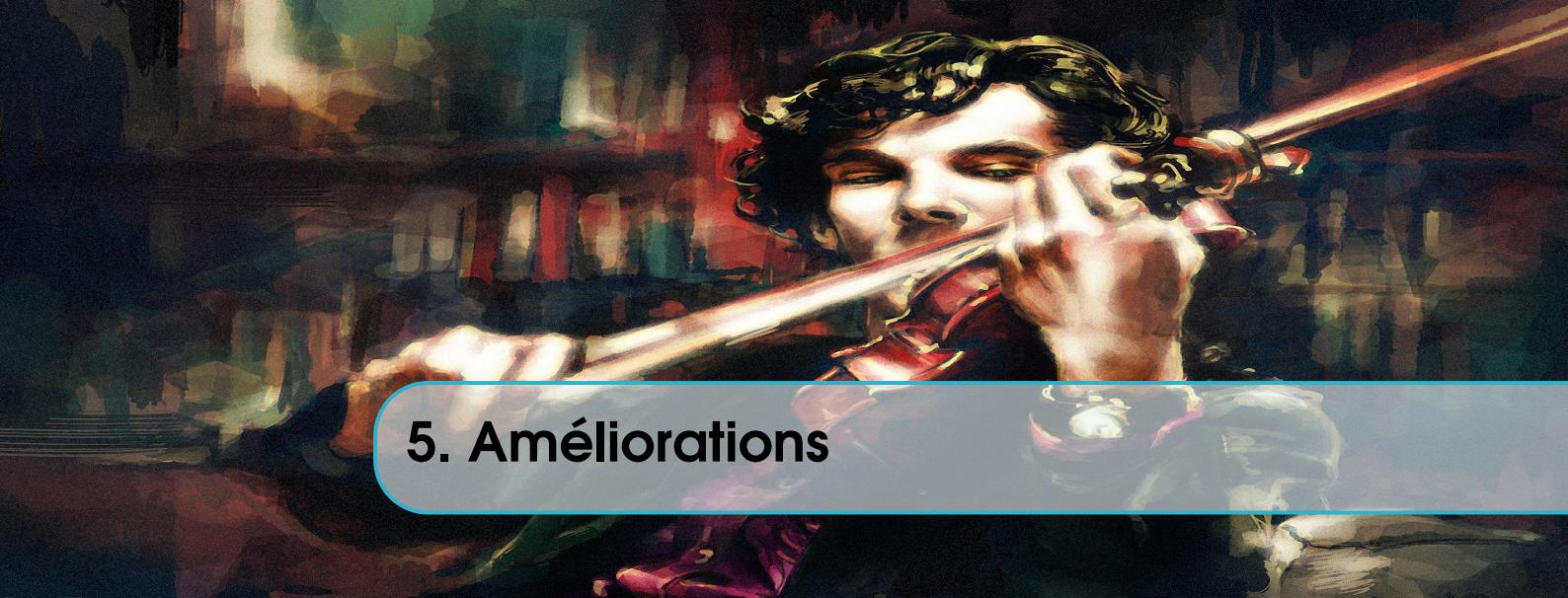
4.3. Ressources complémentaires

Programmation Réseau et Système d'exploitation 1

Ce projet propose une version du jeu Sherlock Holmes

Main Page	Data Structures	Files
server		
server.c File Reference		
<p>server soutenant le jeu Sherlock Holmes. On exécute le server une fois et 4 clients afin de commencer un jeu. Ce jeu nous permet d'aborder les notions fondamentales de la programmation réseau. More...</p> <pre>#include <stdio.h> #include <stdlib.h> #include <string.h> #include <unistd.h> #include <sys/types.h> #include <sys/socket.h> #include <netinet/in.h> #include <netdb.h> #include <arpa/inet.h></pre>		
Data Structures		
<pre>struct _client { Objet client se connectant au serveur. More...</pre>		
Functions		
<pre>void error (const char *msg) { Fonction de gestion d'erreur. More... } void melangerDeck () { Fonction de mélange du deck de cartes afin de les distribuer. More... } void createTable () { Fonction de création de la table de carte qui est une variable globale. More... } void printDeck () { Fonction d'affichage du deck'. More... } void printClients () { affiche le numéro, adresse IP et numéro de port de chaque client More... } int findClientByName (char *name) { Fonction d'affichage du deck'. More... } void sendMessageToClient (char *clientip, int clientport, char *mess) { Fonction d'affichage du deck'. More... } void broadcastMessage (char *mess) { Permet d'envoyer un message à l'ensemble des clients connectés sur le server. More... } int main (int argc, char argv[]) { Fonction principale. }</pre>		
Variables		
<pre>struct _client tcpClients [4] { int nbClients int fsmServer int idDemande int guiltySel int joueurSel int objetSel int deck [13] ={0,1,2,3,4,5,6,7,8,9,10,11,12} int tableCartes [4][8] int banned [4] ={0,0,0,0} char nomcartes [] int joueurCourant char chatserver [256] char tab_texte [8][256] int i = 0 }</pre>		

FIGURE 4.2 – Documentation créée avec le logiciel doxygen



5. Améliorations

5.1 Mise à jour automatique du serveur

Nous avons constaté que l'une des failles de ce jeu était que si en cours de partie un joueur décide d'abandonner (pour une raison qu'on ignore) le jeu plante... Nous avons donc décidé de mettre à jour automatiquement la liste des joueurs, lorsqu'un joueur décide de quitter la partie. Ainsi, la partie recommence et le serveur redistribue aléatoirement les cartes (dans le cas où la partie avait déjà commencé).

5.1.1 Rajout dans le protocole

Si vous souhaitez quitter la partie c'est très simple, il vous suffit de cliquer sur le bouton "quitter" (5.1) qui se trouve au milieu de l'écran SDL à n'importe quel moment de la partie, que ce soit en milieu de jeu ou pas.

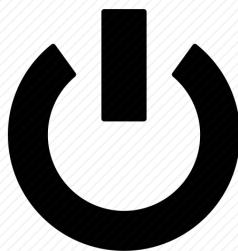


FIGURE 5.1 – Bouton quit

Après avoir pressé le bouton, la situation devient rapidement irréversible... Le client envoie un message au serveur sous la forme "Q iDjoueurCourant", le serveur reçoit le message supprime le joueur courant de la partie en l'enlevant de la liste des joueurs, récupère les cartes et remplace les 3 joueurs restants dans la configuration "d'attente" c'est-à-dire la configuration précédant le début d'une partie... Et tout cela en même pas une seconde. Il faut donc être très vigilant avec l'utilisation de ce bouton, à n'utiliser qu'en cas d'extrême urgence !!

5.1.2 Quels sont les traitements à réaliser ?

Il faut savoir que le traitement se fait uniquement côté serveur, qui doit re-initialiser la partie et remplacer les joueurs. Regardons plus en détail le code associé à la gestion de l'arrêt d'un joueur.

```

case 'Q':
    sscanf(buffer,"Q %d",&idDemande);
    nbClients = nbClients-1;
    nt z = 0;
    for (int i = 0; i < nbClients+1; ++i){
        if (i != idDemande){
            strcpy(temp[z].ipAddress, tcpClients[i].ipAddress);
            strcpy(temp[z].name, tcpClients[i].name);
            temp[z].port = tcpClients[i].port;
            z++;
        }
    }
    printClients();
    initialiser(tcpClients);
    printClients();
    for (int i = 0; i < z; ++i){
        strcpy(tcpClients[i].ipAddress, temp[i].ipAddress);
        strcpy(tcpClients[i].name, temp[i].name);
        tcpClients[i].port = temp[i].port;
    }
    sprintf(reply,"L %s %s %s %s",
    tcpClients[0].name,
    tcpClients[1].name,
    tcpClients[2].name,
    tcpClients[3].name);
    broadcastMessage(reply);
    sprintf(reply,"Q %d",z);
    broadcastMessage(reply);
    fsmserveur=0;
    break;
}

```

Listing 3 – Gestion de l’arrêt d’un joueur

Dès les premières lignes, on diminue le nombre de joueurs de 1, on recopie les joueurs restants dans un tableau temporaire nommé *temp* et on initialise la liste des Clients via la ligne *initialiser(tcpClients)*. Ensuite, on recopie les éléments du tableau *temp* qui contient les 3 structures des clients restants. Le fait de recopier les clients restants sur un tableau temporaire permet de conserver l’ordre d’arrivée des clients.

Pour finir, on envoie un message à l’ensemble des clients qui correspond à la liste des clients actuels, permettant à ces derniers de les lister sur leur tableau. La file d’attente reprend puisque nous nous retrouvons à nouveau à l’état *fsmserveur = 0*, les trois joueurs doivent donc attendre la connexion d’un quatrième joueur avant de reprendre une nouvelle partie.

5.2 Chat global

5.2.1 Contexte

Quoi de mieux qu’un jeu multi-joueurs en ligne avec un chat global ? Une de nos idées innovatrice qui a émergé était de créer un chat au sein du jeu, afin que les joueurs puissent communiquer librement et discuter au cours d’une partie. Nous avons donc décidé de réservé un espace au sein de la surface SDL en bas à droite

5.2. Chat global

de cette dernière pour le chat. Le joueur Lambda peut se connecter au chat à chaque fois qu'il clique sur le bouton correspondant au chat.

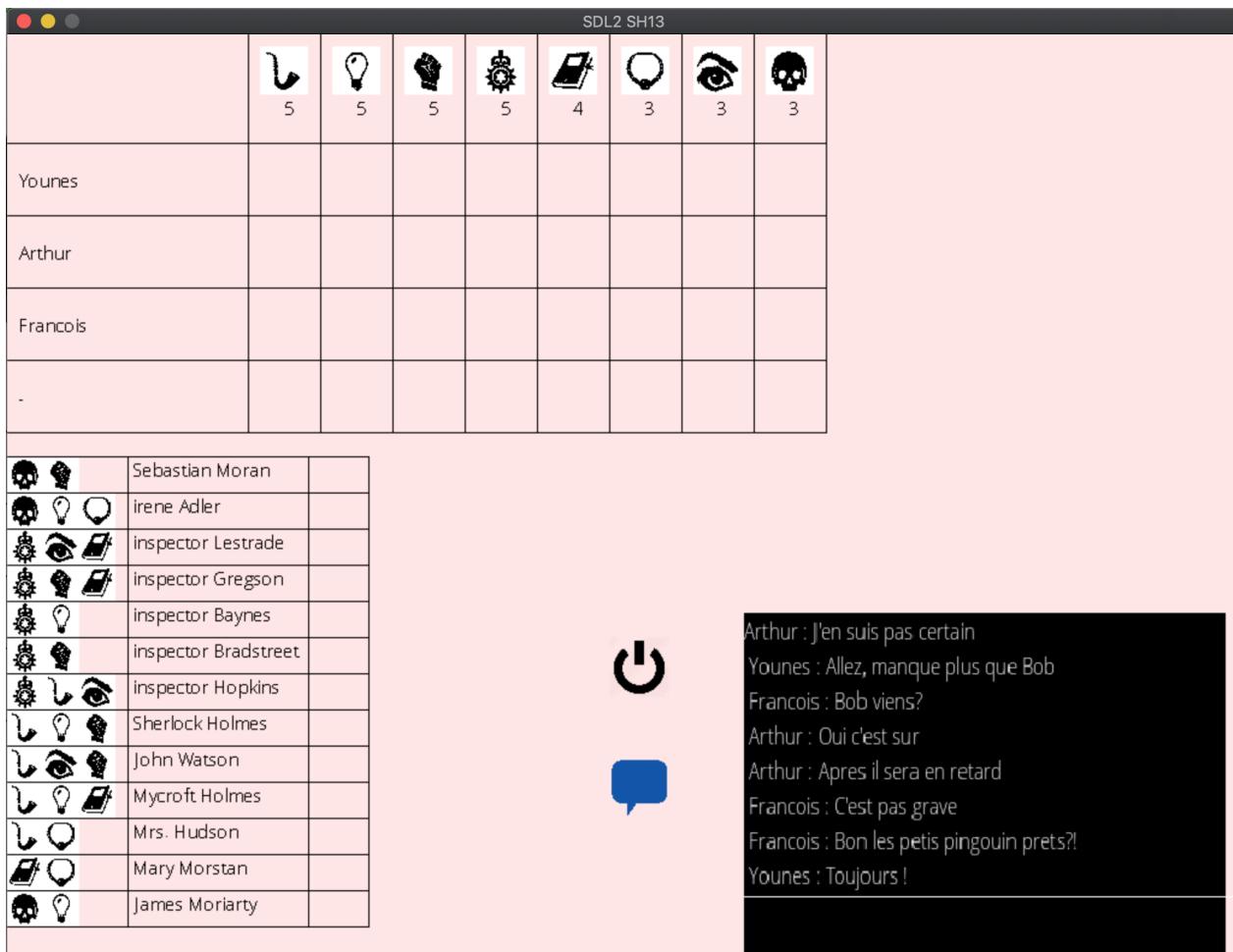


FIGURE 5.2 – SDL chat

5.2.2 Comment cela se traduit-il sur le protocole ?

Les clients et le serveur doivent communiquer afin de comprendre que le client a envoyé un message destiné au chat global. Selon notre protocole, ces derniers communiqueront via un message du type "Z", le client envoie au serveur un message du type "Z messageauchat iDjoueur", le serveur le traite comme suit : "**Je dois ajouter au chat courant contenant les 8 derniers messages du chat le message que je viens de recevoir venant du joueur iDjoueur**". Après reçu le message, le serveur met à jour le chat sous forme d'un unique texte et l'envoie à l'ensemble des clients sous la forme "Z chatActuel" avec chatActuel une chaîne de caractères contenant les 8 derniers messages (avec la provenance des messages) séparés tous d'un saut à la ligne, la variable correspondant à celle-ci est nommée *textecourant* dans le programme. Le client aura juste à afficher cette chaîne de caractère sans se soucier des sauts à la ligne.

5.2.3 Sur le code ?

La difficulté de ce chat est de pouvoir afficher le message et ajuster son affichage en fonction de sa taille. Le traitement se fait des deux côtés ; du côté serveur et du côté client. Une fois qu'un client envoie un message sur le chat, il envoie un message au serveur avec le message correspondant comme précisé précédemment. Nous avions commencé par créer une variable globale qui est un tableau contenant les 8 derniers messages et le serveur remplissait et renvoyait au fur et à mesure ce tableau aux clients présents dans la partie. Il

s'agit d'une méthode très coûteuse et qui implique énormément d'utilisation de boucles afin de parcourir les différents tableaux. C'est ce pourquoi nous avons décidé d'opter pour une autre solution qui est d'écrire sur une chaîne de caractère les 8 derniers messages reçus par le serveur séparés chacun d'entre eux par un saut à la ligne, synonyme du passage au message suivant. Cette méthode s'est avérée très efficace notamment pour l'affichage (via TTF et SDL text render) mais également pour coder les messages afin d'envoyer des messages privés (abordé au point suivant). La documentation assez complètes concernant la manipulation des SDLText ainsi que TTF sont disponibles sur le net.

```
// Cas où la partie n'a pas commencé:

case 'Z':
    sscanf(buffer,"Z %d %[^\\n]s",&idDemande, chatserver);
        sprintf(tab_texte[i], "%s : %s", tcpClients[idDemande].name
i++;
if (i > 7) {
    i = 0;
}
for (size_t j = 0; j < i; j++) {
    sprintf(reply, "Z %s %d", tab_texte[j], i);
    broadcastMessage(reply);
}
break;

// Cas où la partie a commencé (fsmServeur == 1)

case 'Z':
    sscanf(buffer,"Z %d %s %s", &idDemande, chatserver);
    sprintf(chatserver, "%s : %s", tcpClients[idDemande].name, chatserver);
        sprintf(tab_texte[i],"%s",chatserver);
        i++;
if (i > 7) {
    i = 0;
}
for (size_t j = 0; j < i; j++) {
    sprintf(reply, "Z %s %d", tab_texte[j], i);
    broadcastMessage(reply);
}
break;
```

Listing 4 – Gestion de l'arrivée d'un message côté Serveur

Côté client, le code est plus long et plus complexe, nous vous invitons à analyser directement le code via le lien [github](#).

5.3. Chat - message privé

5.3 Chat - message privé

5.3.1 Contexte

Nous voulions pousser l'idée du chat à l'extrême, et faire en sorte qu'à chaque fois qu'un joueur sur le chat souhaite envoyer un message privé à un autre joueur, il lui suffit d'écrire un "@" suivi du nom du joueur destinataire. L'ensemble des joueurs ne verront pas ce message privé mais ils verront des "*" sauf le joueur à qui le message est destiné. Aucun changement n'est réalisé dans le protocole, le texte est directement traité par le client lorsqu'il reçoit le message du serveur "Z chatcourant".

5.3.2 Comment se traduit-il au niveau du code ?

Comme l'ensemble des messages du chat est stocké dans la variable "texte courant" nous parcourons cette chaîne caractère afin de détecter un "@". Dès qu'un "@" est détecté, on récupère le nom qui est suivi par cet "@" jusqu'à s'arrêter au prochain espace, synonyme du début de message.

Si le nom détecté correspond à notre propre nom (ici, gName) alors on affiche le message jusqu'au prochain saut de ligne, sinon l'intégralité du message privé est remplacé par des "*". Afin de comparer des chaînes de caractères (notamment pour le nom), l'utilisation de la fonction strcmp() est indispensable (des documentations très complètes sont disponibles sur internet concernant les fonctions de manipulations de chaînes de caractères en C).

Il faut savoir que cette manipulation se fait au niveau local, c'est à chaque joueur de traiter le texte qu'il reçoit et le "crypté", le serveur n'a rien avoir dans cette histoire. Voici le code associé (sur sh13.c) :

```

case 'Z':
    sscanf(gbuffer, "Z %[^Z]s", texte_courant);
    char texte_temporaire[100];
    int compteur = 0;
    for(int r = 0; r<8; r++){
        while(texte_courant[compteur] != '@'){
            compteur++;
            if (compteur > strlen(texte_courant))
            {
                break;
            }
        }
        if (texte_courant[compteur] == '@')
        {
            char nom_temporaire[40];
            int p = compteur+1;
            int q = 0;
            strcpy(texte_temporaire, texte_courant);
            while(texte_temporaire[p] != ' '){
                nom_temporaire[q] = texte_temporaire[p];
                q++;
                p++;
            }
            if (strcmp(gName, nom_temporaire) != 0)
            {
                while(texte_courant[p] != '\n'){
                    texte_courant[p] = 42;
                    p++;
                }
            }
            compteur = p;
        }
    }
    compteur = 0;
}

```

Listing 5 – Gestion de l’arrivée d’un message

La variable `compteur` correspond à l’indice que nous utilisons afin de parcourir le texte courant (texte contenant les 8 derniers messages du chat). La boucle s’arrête si nous dépassons la taille du texte ou bien si nous rencontrons le caractère "@" condition nécessaire afin d’envoyer un message privé à un joueur. Une fois le caractère @ rencontré on récupère les lettres qui suivent jusque rencontrer un espace. Si le nom récupéré correspond à son propre nom, alors il ne fait rien, sinon il remplace toute la suite du message par des "*" jusque rencontrer un saut à la ligne.

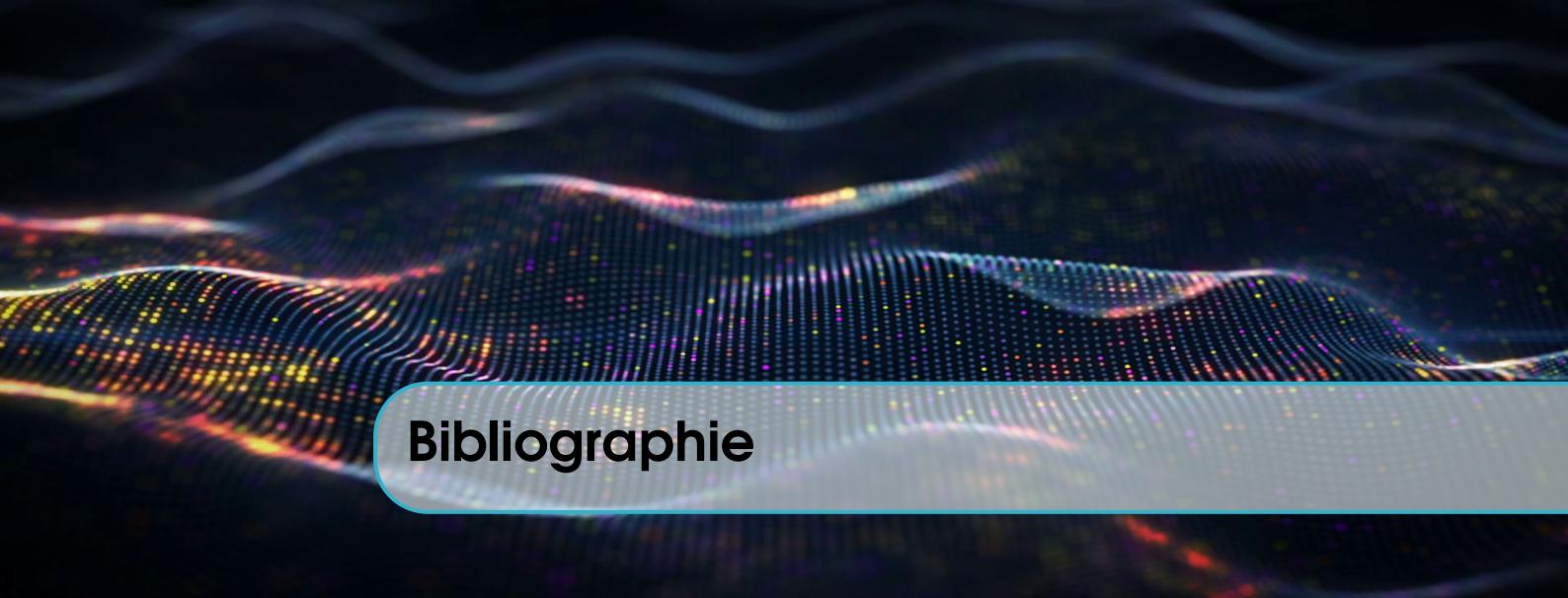


6. Conclusion

Nous avons pris beaucoup de plaisir à réaliser l'intégralité de ce projet. Le modèle de cours/TP proposé nous convenait parfaitement, et le travail en autonomie nous a forcés à prendre des décisions. Cela nous a permis d'apprendre à programmer en deux sur un projet d'une taille conséquente. Nous pouvons dire que nous avons réussi à implémenter avec succès le jeu SH13, et ce jeu nous a permis de comprendre les notions fondamentales de la programmation réseau, qui nous seront certainement utiles lors de notre future carrière. Nous avons poussé les fonctionnalités du jeu à l'extrême, nous amenant à approfondir d'autres notions annexes à la programmation réseau.

D'autres améliorations sont envisageables, et nous aurions pu les aborder si nous avions eu plus de temps. Proposition d'améliorations :

- Scrolling pour le chat
- Notifications sonores pour le chat et pour le déroulement de la partie.
- Thread côté serveur permettant le traitement de plus de joueurs, plus rapidement



Bibliographie

- [1] (4 fév. 2019). Le réseau de zéro, adresse : <https://openclassrooms.com/fr/courses/1561696-les-reseaux-de-zero/3199462-ils-en-tiennent-une-couche-osi-et-tcp-ip> (visité le 14/03/2019).
- [2] (4 fév. 2019). LISTEN(2) Linux Programmer's Manual, adresse : <http://man7.org/linux/man-pages/man2/listen.2.html> (visité le 14/03/2019).