Anaëlle
Agathe
Matthieu

SAÉ 2.03

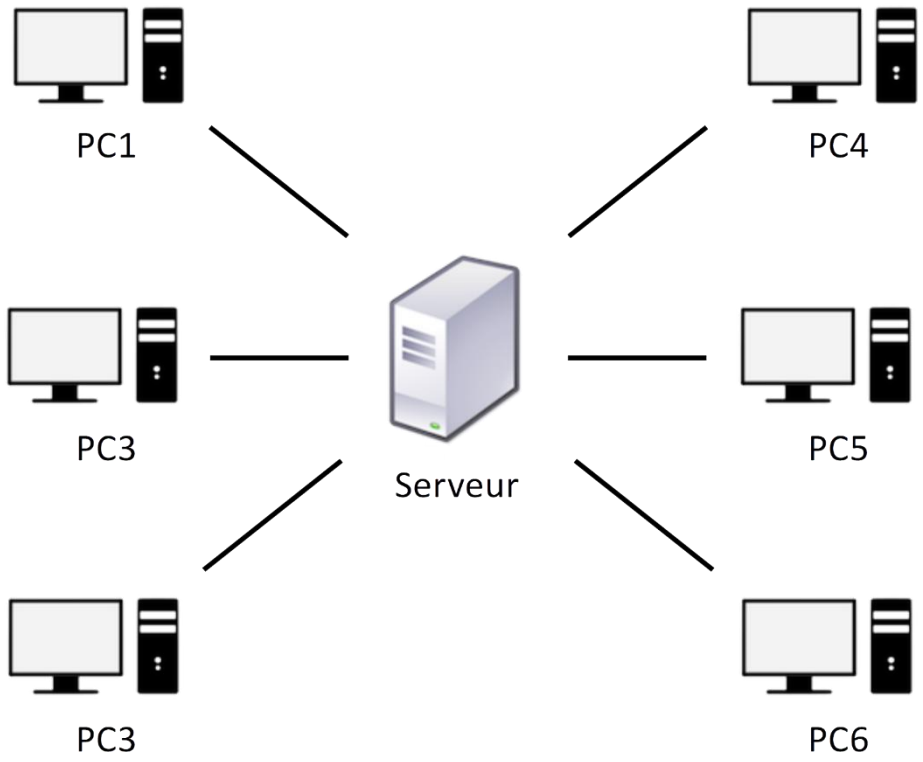
Installation de services réseaux

Table des matières

Présentation

Ce projet consiste en la création d'une application de chat instantané fonctionnant avec un système client/serveur sur un réseau local.

Ce système peut se modéliser de la manière suivante :



L'intérêt étant, chacun se connecte à un serveur servant de nœud central. De cette façon, on peut connecter autant de clients que l'on veut (dans la limite du nombre de postes sur le réseau) au serveur qui traite chaque demande.

Ce concept facilite également l'hébergement : on peut mettre le programme du serveur sur une machine prévue à cet effet, consommant peu et étant allumée en continu. Chaque poste d'utilisateur peut ensuite se connecter quand il le veut.

La communication s'effectuant sur un réseau local, le client et le serveur utilisent des sockets fonctionnant grâce au protocole TCP.

Contraintes

On nous a imposé les contraintes suivantes :

- Le client et le serveur doivent fonctionner dans un environnement Linux
- Le serveur doit pouvoir accepter plusieurs clients à la fois et gérer chacun d'entre eux
- Le serveur doit pouvoir envoyer des messages au même titre qu'un client

Nous avons donc pris la décision de dépasser le sujet et de supporter à la fois Linux et Windows.

Quant au multi-client, il est supporté et il est même possible de se connecter à un serveur hébergé par un système d'exploitation différent que celui du client.

Répartition des tâches

Nous nous sommes réparti les tâches de la manière suivante :

- Cœur du client : Anaëlle
- Cœur du serveur : Agathe
- En-têtes inter-systèmes, rapport, nettoyage et correction des codes, documentation, fonctions utilitaires : Matthieu

L'intérêt de cette répartition est de pouvoir à la fois créer les deux parties du logiciel en parallèle, mais également de développer leurs fonctions communes et de gérer les détails permettant d'optimiser le temps passé sur le projet.

Crédits additionnels :

- https://codes-sources.commentcamarche.net/source/10611-fonctions-kbhit-et-getch-sous-linux-doit-marcher-avec-tout-os-a-base-d-unix-freebsd-etc
- http://tvaira.free.fr/bts-sn/reseaux/cours/cours-programmation-sockets.pdf
- StackOverflow

Voir le code source

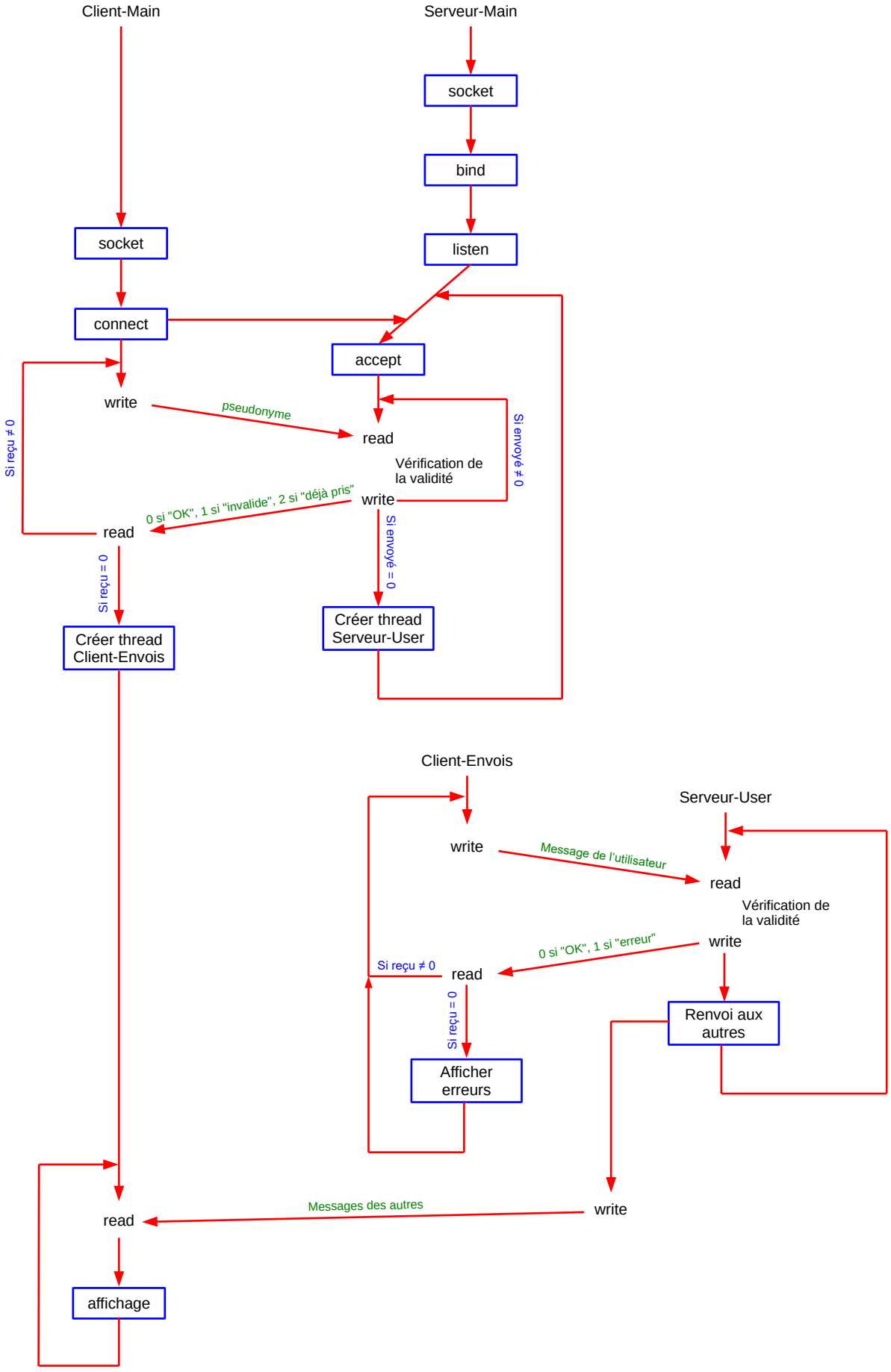
Le code source écrit dans le cadre de ce projet est joint au présent rapport.

Il était demandé de le mettre en annexe, mais celui-ci est trop long pour pouvoir y tenir, en plus d'être divisé en 8 fichiers pour une meilleure organisation.

Certains extraits sont cependant présents pour mieux expliquer certains points.

Modélisation du fonctionnement

Le fonctionnement du client et du serveur peut se modéliser par le schéma suivant :



La modélisation du fonctionnement du projet permet de noter plusieurs éléments :

- Certaines tâches ont besoin d'être exécutées en parallèle, donc le client a deux fils d'exécution (« threads »), et le serveur en a au minimum deux, au maximum le nombre d'ordinateur présent sur le serveur plus le thread de connexion.

- Un utilisateur se connectant au cours d'une conversation ne recevra pas les messages antérieurs à son arrivée.
  Fun fact : l'un des premiers systèmes de communication à distance par internet, introduit en 1988 dénommé « IRC » (« Internet Relay Chat ») avait la même contrainte.

- Un client souffrant d'une erreur ne signifie pas forcément une déconnexion, mais une invalidité du message envoyé.

Ces éléments sont importants et servirons par la suite.

Notez que cette modélisation est la modélisation du problème initial. Le serveur a un thread de plus que sur la modélisation, qui lui permet d'envoyer des messages comme un utilisateur comme requis par les contraintes.

Protocoles et conventions

Il a fallu définir une façon de formatter les messages échangés. On appelle cela un protocole : c'est l'encodage d'informations sous la forme d'octets en suivant des conventions précises.

Pour les codes d'erreurs, il nous suffit d'envoyer un seul octet contenant la valeur de notre choix.
Or, pour le texte, comment faire ?

L'UTF-8 est un standard d'encodage permettant l'écriture de tout caractère possible et imaginable, grâce à des conventions précises.

Le but ici n'étant pas de les détailler, nous vous invitons à lire l'article Wikipédia sur le sujet :
https://en.wikipedia.org/wiki/UTF-8

Le terminal Linux utilise ce standard pour gérer ses entrées et sorties de texte. De plus, la compilation de chaines de caractères avec le compilateur GNU les interprète dans ce format également.

Nous avons donc choisi de transmettre notre texte encodé sous la forme d'une chaîne d'octets suivant le standard UTF-8 terminée par un caractère nul de terminaison.

Ainsi, la phrase « j'ai bien mangé » serait transmise sous la forme de la chaîne d'octets suivante (valeurs écrites en hexadécimal):

j	'	a	i		b	i	e	n		m	a	n	g	é	
6A	27	61	69	20	62	69	65	6E	20	6D	61	6E	67	C3 A9	00

Organisation des traitements

Vérification des pseudonymes

Le client va devoir répondre à certains critères avant de pouvoir rentrer dans la messagerie. En effet, le serveur attendra le pseudo du client après la connexion de celui-ci. Il doit respecter les conditions suivantes :

- Il doit contenir seulement des lettres ou des nombres. Les autres caractères ne sont pas acceptés. Cela permet aussi de ne pas confondre les messages d'un client avec ceux du serveur (qui utilise des crochets). Si cette condition n'est pas respectée, Le serveur renverra 1. C'est une erreur qui va s'afficher et prévenir le client de l'incompatibilité de son pseudo choisi.

```
for(int a = 0; a < lu-1; a++) {
    if (!((idRecu[a] >= 'a' && idRecu[a] <= 'z') || (idRecu[a] >= 'A' &&
            idRecu[a]<= 'Z') || (idRecu[a] >= '0' && idRecu[a] <= '9'))) {
        verif = 1;
        break;
    }
}
```

- Il doit être unique. Si le pseudo est déjà utilisé, le serveur renverra 2, ce qui va avertir le client dudit problème. Ceci permet d'éviter les doublons afin de permettre une discussion normale sans que deux personnes du type « Matthieu » ne viennent vous embêter.

```
for(int b = 0; b < nb_id; b++) {
    if(strcmp(Nom_Id[b].data(), idRecu) == 0) {
        verif = 2;
        break;
    }
}
```

Si ces conditions sont respectées, le serveur renvoie 0 au client, ce qui lui signale qu'il peut envoyer des messages dans le chat.

Les déconnexions

Après la connexion et l'acceptation d'un client, le serveur va automatiquement lui créer un thread dédié pour gérer ses messages.

Une boucle est créée et permet de conserver le socket associé au client tant qu'il reste connecté au serveur.

Lorsque le serveur reçois un message de sa part, il vérifiera les éventualités suivantes :

- Une déconnexion inattendue du client

```
sprintf(messageGlobal, "<- %s a été déconnecté.", pseudo);
br = true;
```

- Une fermeture volontaire du programme par le client

```
sprintf(messageGlobal, "<- %s est partit.", pseudo);
br = true;
```

Si l'un de ces problèmes survient, le serveur renvoie alors une phrase descriptive sur le chat et ferme le socket correspondant.

Autrement, il enverra le message reçu à tous les utilisateurs grâce à la fonction `broadcastAll`.

```
broadcastAll(messageGlobal, socketDialogue);
```

Celui-ci s'affichera également sur la fenêtre du serveur.

Les fonctions

broadcastAll

La fonction `broadcastAll` va permettre d'envoyer des messages à tous les clients connectés. Il va prendre en paramètre le numéro du socket de l'émetteur (le client qui envoie le message) afin d'éviter de lui renvoyer, puisqu'il en est à l'origine. Il va ensuite parcourir tous les sockets disponibles et va leur envoyer le message voulu. Si l'on veut envoyer le message à tous sans exception, on peut passer « 0 » au lieu du numéro de socket émetteur.

```
void broadcastAll(const char* str, SOCKET except) {
      int nbSockets = sockets.size();

      for(int i = 0; i < nbSockets; i++) {
            if(sockets[i] != except) {
                  int ecrits = send(sockets[i], str, strlen(str)+1, 0);
                  gererErreurs(ecrits, sockets[i], "envoi", true, false);
            }
      }

      log("%s\n", str);
}
```

leave

La fonction `leave` va permettre au client de partir en fermant son socket. Son numéro de socket est pris en paramètre. Alors, on va parcourir tous les sockets existants et une fois que l'on a trouvé celui correspondant, on peut le retirer de la liste lui et son pseudonyme attitré, puis refermer celui-ci.

```
void leave(SOCKET socket) {
    for(unsigned int i = 0; i < nbUsers; i++) {
        if(sockets[i] == socket) {
            sockets.erase(sockets.begin() + i);
            Nom_Id.erase(Nom_Id.begin() + i);
            nbUsers--;
            gererErreurs(closesocket(socket), socket, "fermeture", true, false);
            break;
        }
    }
}
```

log

La fonction `log` permet d'enregistrer efficacement les données. Elle va afficher l'heure avant un message formatté, puis sauvegarder le tout dans un fichier de journalisation.

Problèmes rencontrés

Différentiation des messages reçus par le client

Exposition du problème

Le client peut recevoir deux types de messages : un code de retour, ou un texte à afficher.

Or, le système de socket ne fait pas la différence entre les deux par lui-même.

Donc, il a fallu élaborer un système permettant de trier ceux-ci.

Notre solution

Nous avons choisi de créer un espace global retenant les messages reçus, afin que le thread correspondant puisse le traiter selon certains critères.

Le seul thread qui reçoit les messages est le thread principal, qui va vérifier la longueur de ceux-ci.

C'est simple : un code de retour est obligatoirement composé d'un seul et unique octet, alors qu'un texte à afficher est long d'au moins deux octets (un caractère et le caractère nul de terminaison).

Alors, le thread principal affiche le message sous forme de texte si cette dernière condition est respectée, et, dans le cas contraire, va le stocker dans l'emplacement global mentionné afin d'être traité par le thread d'envois.

Compatibilité Windows

La comptabilité entre les systèmes n'est pas toujours simple, beaucoup de travail a dû être fait pour assurer celle-ci.

Les sockets

Les sockets ont été les plus simples à adapter.

Le cours de Thierry VAIRA sur les sockets propose l'en-tête suivant (modifié légèrement par nos soins) :

```
#pragma once

#ifdef WIN32 /* si vous êtes sous Windows */

#include <winsock2.h>
typedef int socklen_t;

#elif defined (linux) /* si vous êtes sous Linux */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h> /* close */
#include <cstdlib> /* exit */
#include <netdb.h> /* gethostbyname */
#define INVALID_SOCKET -1
#define SOCKET_ERROR -1
#define closesocket(s) close(s)
typedef int SOCKET;
typedef struct sockaddr_in SOCKADDR_IN;
typedef struct sockaddr SOCKADDR;
typedef struct in_addr IN_ADDR;

#else /* sinon vous êtes sur une plateforme non supportée */

#error not defined for this platform

#endif
```

Cet en-tête est très complet, et grâce à celui-ci, la syntaxe d'utilisation des sockets pourra être compilée à la fois sur Linux et Windows.

Cette simplicité s'explique par le grand nombre de fonctions communes entre l'API de sockets Linux et Windows, évitant ainsi d'avoir à recréer des fonctions. De simples déclarations d'alias suffisent.

Les threads

Les threads furent particulièrement difficiles à rendre compatible entre Linux et Windows.

Linux utilise l'API « pthread », qui gère des threads dans une structure `pthread_t`, qui contiennent les différentes propriétés de celui-ci. Une fonction d'entrée d'un thread de cette librairie prends en entrée un pointeur quelconque, et en retourne un également.

Windows utilise l'API « Win32ProcessThreads », qui gère ses threads par des identifiants `HANDLE`. Une fonction d'entrée d'un thread de cette librairie prend en entrée un double mot (`DWORD`) de 8 octets, et en retourne un également.

Ainsi, il a fallu adapter ces contraintes afin de faire un parallèle entre pthread et Win32ProcessThreads.

La solution retenue a été de créer des fonctions reproduisant le comportement de pthread et empruntant leur nom sous Windows, qui en interne agissent de manière adaptée à Win32ProcessThreads.

Nous avons donc écrit un en-tête effectuant ce travail grâce à la documentation des deux librairies.

En voici un extrait (version complète avec le reste du code) :

```
inline int pthread_create(pthread_t *thread,
                                        const pthread_attr_t *attr,
                                        void *(*start_routine)(void *),
                                        void *arg)
{
        if(thread == NULL)
                return -1;

        thread->start_routine = start_routine;
        thread->arg = arg;
        thread->result = NULL;

        DWORD stackSize = (attr == NULL) ? 0 : attr->__stacksize;
        thread->handle = CreateThread(
                                NULL,                                   // Attributs de sécurité par défaut
                                stackSize,                              // Taille de la pile
                                (LPTHREAD_START_ROUTINE)start_routine,  // Point d'entrée du thread
                                arg,                                    // Argument à passer à ladite fonction
                                0,                                      // Flags de création
                                &thread->tid);                          // Pointeur vers la variable où écrire
                                                                           l'identifiant du thread

        if(thread->handle == NULL)
                return -1;

        return 0;
}
```

La structure `pthread_t` et ses variantes ont été recréées au préalable, la fonction imite donc parfaitement son comportement sous Linux.

Au début de cette section, nous avons mentionné la différence de type de la valeur d'entrée et de retour entre les threads Linux et Windows : Linux prend et renvoie un pointeur, alors que Windows prends et envoie un double mot.

Ceci n'est en réalité pas un problème : un pointeur fait, en réalité, sur un système 64 bits, la même taille qu'un double mot. Sur un système 32 bits, ceux-ci font la taille d'un mot simple. Grâce au casting, il est donc possible de faire passer nos pointeurs passés pour des double mots sans perdre en précision ou en informations.

La console

La console aussi dispose de ses problèmes de compatibilité.

Comme dit précédemment, Linux prends ses entrées et sorties de terminal au format UTF-8, que nous avons donc choisit d'utiliser pour encoder notre texte à transmettre entre les clients et le serveur.

Or, Windows utilise par défaut le format Code page 850 : https://en.wikipedia.org/wiki/Code_page_850
Ce format diffère beaucoup de l'UTF-8. Convertir de l'un à l'autre n'est pas facile.

Fort heureusement, l'API Windows possède deux fonctions pour remédier (partiellement) à ce problème.

```
SetConsoleOutputCP(CP_UTF8);
```

Grâce à cette fonction, les sorties de texte de la console Windows sont maintenant faits en UTF-8.

Mais qu'en est-il des entrées de texte ? Windows possède bien une fonction pour changer l'encodage des entrées :

```
SetConsoleCP(CP_UTF8);
```

Mais cette procédure casse le fonctionnement des fonctions de capture d'appui de touche, les faisant boucler à l'infini voire causant un plantage.

Nous avons donc dû convertir « à la main » le Code page 850 vers l'UTF-8, grâce à une fonction écrite par nos soins :

```
const char* convTable[] = {
	"Ç", "ü", "é", "â", "ä", "à", "å", "ç", "ê", "ë", "è", "ï", "î", "ì", "Ä", "Å",
	"É", "æ", "Æ", "ô", "ö", "ò", "û", "ù", "ÿ", "Ö", "Ü", "ø", "£", "Ø", "×", "ƒ",
	"á", "í", "ó", "ú", "ñ", "Ñ", "ª", "º", "¿", "®", "¬", "½", "¼", "¡", "«", "»",
	"░", "▒", "▓", "│", "┤", "Á", "Â", "À", "©", "╣", "║", "╗", "╝", "¢", "¥", "┐",
	"└", "┴", "┬", "├", "─", "┼", "ã", "Ã", "╚", "╔", "╩", "╦", "╠", "═", "╬", "¤",
	"ð", "Ð", "Ê", "Ë", "È", "ı", "Í", "Î", "Ï", "┘", "┌", "█", "▄", "¦", "Ì", "▀",
	"Ó", "ß", "Ô", "Ò", "õ", "Õ", "µ", "þ", "Þ", "Ú", "Û", "Ù", "ý", "Ý", "¯", "´",
	NULL, "±", "‗", "¾", "¶", "§", "÷", "¸", "°", "¨", "·", "¹", "³", "²", "■", NULL

};

// Convertit le Code page 850 de Microsoft vers l'UTF-8 pour une intercompatiblité des systèmes d'exploitation
const char* CP850ToUTF8(char chr) {
	if(chr > 0)
		return NULL;

	return convTable[chr + 0x80];
}
```

Capture d’appuis de touches

Nous avons mentionné les fonctions de capture d’appui de touche, et pour cause : celles-ci sont également différentes entre les deux systèmes.

Sur Windows, l’API `conio` donne l’accès aux fonctions `_kbhit()` et `_getch()`, qui respectivement capturent un appui de touche et récupèrent le caractère écrit par celui-ci.

Sur Linux, il faut changer les propriétés du terminal pour désactiver son retour d’écho avec l’API `termios`, puis lire le descripteur du fichier `stdout` pour récupérer le caractère du dernier appui de touche.

Fort heureusement, l’utilisateur MickBad (voir crédits additionnels) a écrit un en-tête permettant de reproduire les fonctions `_kbhit()` et `_getch()` sur Linux. Cette adaptation sortant un peu du sujet, nous nous sommes permis d’utiliser son travail.

Capture de touches en temps réel

Exposition du problème

Un bon travail est non seulement fonctionnel, mais également joli.

C'est avec cette philosophie que nous nous sommes heurtés au problème suivant :

- Deux personnes discutent.

  ```
  Jean: Bonjour les amis !
  Michel: Bonjour jean, ça va ?
  Jean: ▂
  ```

  Sur le client de Jean, il peut voir son nom et un espace vide pour écrire.

- Or, si Michel lui envoie un nouveau message avant qu'il puisse envoyer le sien, cette zone d'écriture doit se déplacer :

  ```
  Jean: Bonjour les amis !
  Michel: Bonjour jean, ça va ?
  Michel: Au fait, as-tu vu mon dernier mail ?
  Jean: ▂
  ```

- De la même manière, si Jean est en train d'écrire…

  ```
  Jean: Bonjour les amis !
  Michel: Bonjour jean, ça va ?
  Michel: Au fait, as-tu vu mon dernier mail ?
  Jean: Oui, ça va, et effectivement je l'▂
  ```

  …et qu'il reçoit un message en même temps…

  ```
  Jean: Bonjour les amis !
  Michel: Bonjour jean, ça va ?
  Michel: Au fait, as-tu vu mon dernier mail ?
  Michel: Parce que bon, ça fait une semaine que j'attends !
  Jean: Oui, ça va, et effectivement je l'▂
  ```

  Sa zone d'écriture doit se déplacer tout en lui permettant de continuer d'écrire et de modifier son texte.

  ```
  Jean: Bonjour les amis !
  Michel: Bonjour jean, ça va ?
  Michel: Au fait, as-tu vu mon dernier mail ?
  Michel: Parce que bon, ça fait une semaine que j'attends !
  Jean: Oui, ça va, et effectivement je l'ai vu
  ```

Cela peut paraitre évident : beaucoup d'applications de chats fonctionnent de cette manière. La différence ici est que notre environnement est un terminal (Linux) ou une console (Windows).

Notre solution

Aucune fonction de la librairie standard C/C++ ne permets de faire une saisie de la sorte qui pourrait être interrompue pour être reprise une ligne plus loin.

Nous avons donc dû créer une alternative, une « fausse » version de la fonction de saisie `std::cin`.

Son code est disponible avec le reste sous le nom de `fakeCin`. Cette fonction fut un véritable défi vis-à-vis de la compatibilité inter-systèmes et des options pour manipuler le terminal ou la console pour effectuer une saisie contrôlable de A à Z.

Estimation de l'adresse IP

Exposition du problème

Héberger un serveur est très simple : il suffit de lancer le programme du serveur et tout est prêt.

En revanche, il faut bien donner notre IP locale aux gens que l'on veut faire se connecter à notre serveur.

Hélas, il n'existe pas de méthode simple pour cela, et exécuter la commande `ifconfig` (Linux) ou `ipconfig` (Windows) n'est pas quelque chose qui vient à l'esprit des non-programmeurs.

Le serveur doit donc essayer de trouver son adresse IP locale à partager.

Notre solution

Grâce à l'API de sockets, nous pouvons trouver notre propre adresse IP :

```
// https://stackoverflow.com/a/122225
bool getMyIP(unsigned char* ip)
{
	char szBuffer[1024];

	if(gethostname(szBuffer, sizeof(szBuffer)) == SOCKET_ERROR)
	{
		return false;
	}

	struct hostent *host = gethostbyname(szBuffer);
	if(host == NULL)
	{
		return false;
	}

	ip[0] = ((unsigned char*)(host->h_addr))[0];
	ip[1] = ((unsigned char*)(host->h_addr))[1];
	ip[2] = ((unsigned char*)(host->h_addr))[2];
	ip[3] = ((unsigned char*)(host->h_addr))[3];

	return true;
}
```

Cette fonction fonctionne bien sur les ordinateurs classiques, elle peut parfois renvoyer la mauvaise adresse si l'ordinateur est connecté à plusieurs périphériques réseau.

Conclusion

Pour conclure, notre application de messagerie instantanée nous aura permis de mettre en application nos connaissances sur les réseaux, et d'approfondir nos capacités à réfléchir et à mettre en pratique le multithreading.