INF 2005 Programmation orientée objet avec C++

Texte 5

Version du 13 janvier 2015

1. Notions sur la programmation parallèle

La programmation parallèle (PP) est un paradigme de programmation qui permet de partitionner un traitement en plusieurs tâches (sous-programmes) élémentaires et d'exécuter plusieurs instructions ou des opérations parallèlement sur des systèmes à plusieurs processeurs ou multicœurs. Ces différentes tâches coopèrent pour réaliser un objectif commun.

Ce paradigme s'est développé étant donné les contraintes qui existaient dans les architectures anciennes. Ces contraintes étaient liées notamment :

- aux problèmes de performance des architectures (la mémoire, la puissance des processeurs, les entrées-sorties, etc.);
- aux problèmes de synchronisation;
- aux restrictions des possibilités de coopération;
- aux difficultés de résoudre rapidement des problèmes à grande échelle.

La programmation parallèle (PP) s'avère la solution idéale pour répondre à ces problèmes rencontrés dans les architectures classiques. Il permet d'accroître et d'améliorer les performances de calcul des systèmes. Il permet non seulement de résoudre rapidement les problèmes qui demandent beaucoup de temps processeur, mais aussi de garantir la résistance aux pannes et à la tolérance aux fautes. De même, ce type de programmation répond aux besoins de performance et de coopération des applications géographiquement réparties.

L'apparition des processeurs multicœurs atteste ce regain d'intérêt pour les architectures parallèles. Nombreux sont les systèmes qui sont actuellement pourvus de technologies multicœurs.

1.1 Intérêt de la programmation parallèle (PP)

La PP est utilisée dans plusieurs domaines, notamment :

- la météorologie et la climatologie;
- l'industrie aéronautique;
- la modélisation des simulations;
- les traitements et l'analyse des données;
- les traitements des images;
- etc.

L'utilisation du calcul parallèle est nécessaire dans tous ces domaines pour diminuer le temps de traitement des données, accélérer les simulations, augmenter l'espace mémoire, etc.

Intégrer le principe de traitements en parallèle en informatique a une conséquence sur la façon de développer les programmes. Les programmes séquentiels (*mono-thread*) ne s'exécutent que sur un seul cœur alors que les programmes parallèles sont multicœurs. Il faut alors les adapter, car l'ajout de processeurs supplémentaires n'améliorera pas automatiquement les performances des applications *mono-threads*. Cette adaptation est du ressort du programmeur qui doit introduire ce nouveau paradigme dans la structuration de son code machine.

Comme autres avantages de la PP, on peut citer entre autres :

- une meilleure utilisation des processeurs, de la mémoire grâce à la multiprogrammation;
- un partage des ressources;
- un meilleur découpage des programmes en tâches;
- une bonne coopération entre les programmes;
- une rapidité d'exécution des programmes sur les machines multiprocesseurs;
- une répartition et une rapidité d'exécution des programmes sur un réseau.

1.2 Les programmes parallèles et les ordinateurs parallèles

Comment peut-on adapter nos anciens programmes à la programmation parallèle? Plusieurs options s'offrent à nous.

Il est possible de paralyser les applications sans récrire totalement le code source de nos programmes. Le plus souvent on utilise les fils ou *threads* de Windows ou ceux de Posix pour faire des programmes parallèles. Mais il existe aussi des solutions pour la migration de programmes séquentiels vers des programmes parallèles.

Une de ces solutions est l'utilisation de l'outil *OpenMP*[1]. Dédié aux langages C, C++ et fortran, *OpenMP* est actuellement utilisé par de nombreux développeurs de logiciels et offre des avantages pour le développement rapide ou la migration des codes sources vers des applications parallèles.

[1] Site web : http://openmp.org/wp/

Un ordinateur parallèle est une machine qui possède plusieurs unités centrales (UC) de traitement, laquelle peut coopérer avec d'autres systèmes et effectuer des traitements en parallèle. Selon la classification de *Flynn*[2], il existe plusieurs types d'ordinateurs munis d'architectures parallèles formées par plusieurs processeurs qui coopèrent aux traitements des applications. On peut les classer selon le type d'organisation du flux de données et du flux d'instructions, illustré à la figure 1.

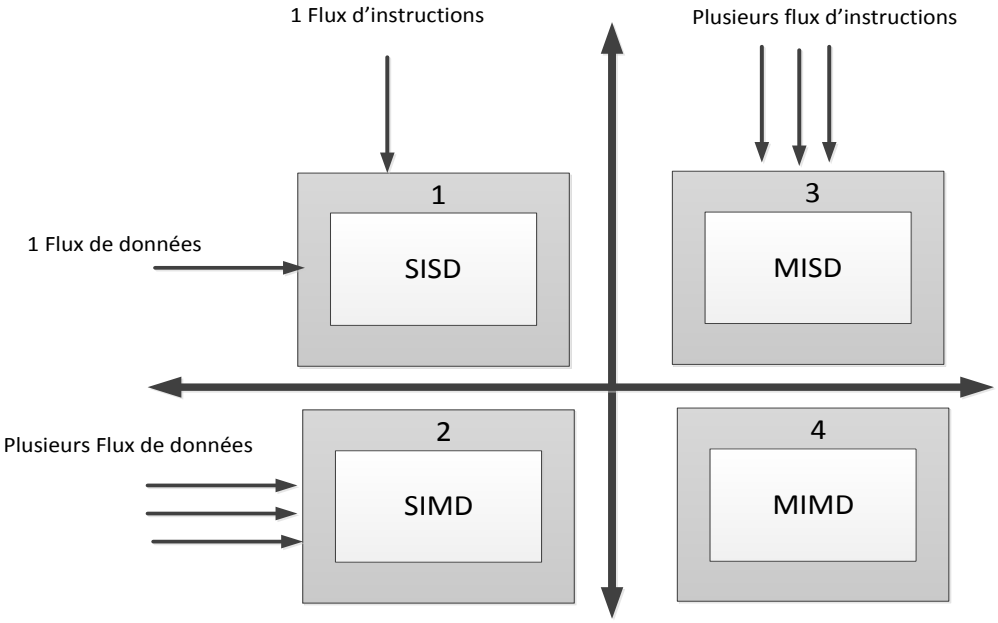

Figure 1 Classification de *Flyn* des ordinateurs.

1. *SISD (Single Instruction on Single Data)* : représente des ordinateurs dont l'architecture est de type *von Neumann.* Ce sont des machines séquentielles dans lesquelles une donnée est traitée par une instruction.
2. *SIMD (Single Instruction on Multiple Data)* : représente des ordinateurs où plusieurs données sont traitées en même temps par un seul flux d'instructions.
3. *MISD (Multiple Instruction on Single Data)* : représente des machines pipelines dans lesquelles une donnée est traitée par plusieurs instructions.
4. *MIND (Multiple Instruction on Multiple Data)* : dans ce type d'architecture les processeurs sont autonomes, la mémoire est partagée et chaque processeur exécute ses propres instructions et agit sur ses données.

[2] Pour des précisions, vous pouvez consulter l'article Wikipédia sur la taxonomie de Flynn (version anglaise).

Notez qu'actuellement les machines sur le marché possèdent dans la majorité des cas plusieurs processeurs et sont capables d'exécuter des tâches en parallèle.

2. Programmation répartie ou distribuée

Les systèmes distribués répondent aux problèmes de la centralisation des ressources qui existent dans les systèmes classiques. Dans les systèmes distribués, les ressources ne sont plus centralisées, mais réparties sur des systèmes parfois homogènes, hétérogènes et mobiles.

2.1 Définition d'un système distribué

Un système distribué est « un ensemble d'ordinateurs indépendants qui se présente aux utilisateurs comme un système unique et cohérent »[3].

Contrairement à un système centralisé, un système distribué est composé de plusieurs systèmes autonomes qui communiquent à travers un réseau ou tout autre dispositif. Ce dernier peut être, par exemple, un intergiciel (*middleware)* qui coordonne et connecte les composants entre eux pour former un système cohérent.

Comme exemple de système distribué, mentionnons les robots footballeurs, les services web comme FTP ou WWW, les agents intelligents, etc. Les ressources distribuées peuvent être plusieurs éléments réseau, comme les ordinateurs, les imprimantes, des fichiers de données, les pages web.

Les avantages d'utiliser des systèmes distribués sont entre autres :

- d'avoir un environnement hétérogène et dynamique qui peut croitre selon les besoins;
- de rendre les ressources facilement accessibles, partageables de façon efficace et contrôler;
- d'avoir un système résistant aux pannes;
- de faciliter les collaborations et les échanges d'information entre les composants du système;
- de profiter des performances de plusieurs processeurs à bas prix qui collaborent et traitent de gros volumes d'information;
- de faire des économies, car, par exemple, sur un réseau, une ressource partagée est presque moins chère qu'une ressource dédiée pour un utilisateur;

[3] Tanenbaum, A.S. et van Steen, M. (2006). *Distributed systems : principles et paradigms* (2e éd.). Upper Saddle River, NJ : Prentice Hall.

- de rendre transparentes les ressources. En effet, pour accéder aux ressources ou pour les partager, l'utilisateur ne doit pas se soucier de leur localisation.

2.2 Les caractéristiques des systèmes distribués

Les systèmes distribués doivent avoir certaines caractéristiques qui sont parfois liées à la nature même de l'environnement où les composants évoluent. Le système doit-être :

- hétérogène, formé avec des machines indépendantes et autonomes. Cela veut dire qu'un système distribué peut être constitué d'un ensemble de systèmes informatiques munis de systèmes d'exploitation différents, chacun ayant leur propre convention de gestion des fichiers;
- ouvert, flexible et facile d'extension;
- disponible en permanence même si certaines parties peuvent être temporairement hors d'usage;
- facile à configurer : le système doit être facile à configurer à partir des composants hétérogènes. Il doit faciliter l'ajout et le remplacement des composants;
- évolutif et garder toujours ses performances.

2.3 Les types de systèmes distribués

Les systèmes distribués peuvent être de divers types, mentionnons entre autres :

- Les systèmes distribués informatiques qui comprennent les systèmes formés par :
  - des grappes (*cluster*s) formées par des machines hétérogènes interconnectées par un même réseau;
  - les grilles de calcul formées par des systèmes hétérogènes qui collaborent. Dans ce cas, les ressources proviennent de plusieurs personnes qui collaborent sous forme d'un environnement virtuel pour partager les données.
- Les systèmes d'information distribués. Ce point concerne l'intégration des composants au sein des entreprises laissant les applications dialoguer directement entre elles. Notamment, c'est le cas des systèmes de traitements et de transactions. Par exemple, un traitement final qui résulte de deux traitements différents sur deux bases de données différentes.
- Les systèmes embarqués distribués où les nœuds du système sont hétérogènes et mobiles en même temps. C'est le cas actuellement des téléphones portables qui forment un environnement de type ad hoc et assurent les échanges, les traitements de données entre composants. On peut citer aussi les systèmes distribués liés aux soins de santé qui sont des dispositifs mis au point pour surveiller le bien-être des individus. Les médecins peuvent ainsi surveiller les malades à distance.

2.4 Les environnements distribués

Parmi les environnements disponibles, mentionnons :

- DCE *(Distributed Computing Environment*) qui est le plus répandu. Il est produit par OSF *(Open Systems Foundation)* et supporte les RPC (appel de procédure à distance). Il permet surtout de résoudre le problème de l'interopérabilité des composants dans les systèmes distribués hétérogènes.
- DCOM *(Distributed Component Object Model)* est un modèle évolué de COM et est adapté pour les systèmes Microsoft. Il permet de partitionner une application en plusieurs composants distribués qui peuvent être exécutés séparément de façon transparente sur des systèmes distants. À cet effet, l'utilisateur n'a pas à se soucier de la localisation d'une application.
- JAVA RMI *(Java Remote Method Invocation)* est spécifique du langage JAVA. C'est une interface de programme qui permet de faire des appels de méthodes à distance. Son fonctionnement est proche de celui de CORBA.
- CORBA. Actuellement, CORBA est très utilisé dans les entreprises pour développer les systèmes distribués. C'est une architecture logicielle souple qui fait partie du projet ODP-RM *(Open Distributed Processing Reference Model)* dont l'objectif premier était de mettre en place une norme d'architecture distribuée. CORBA est actuellement un standard qui est implémenté dans plusieurs applications à travers les outils comme les utilitaires communs *(CORBA facilities*), les services objet communs (*CORBA services*), les interfaces de domaines (*Domain interfaces*).

Ces environnements de systèmes distribués se fondent sur deux modèles : le modèle client-serveur et le modèle avec les intergiciels (*middleware*).

2.4.1 Le modèle client-serveur

Les architectures clients-serveurs sont utilisées dans beaucoup d'applications distribuées. Les intervenants sont divisés en deux groupes : d'un côté les clients et de l'autre les serveurs. Un programme client est un processus demandeur de ressources. Il adresse une requête de ressources à un serveur et reste en attente de la réponse. Quant au serveur, c'est une application pourvoyeur de ressources. Il reste toujours en attente des requêtes de demandes de ressources des clients. L'interaction entre les clients et les serveurs s'effectue souvent par des applications comme les sockets.

2.4.2 Le modèle avec les intergiciels (middleware)

Un intergiciel (*middleware)* est une couche logicielle qui fait la liaison entre les applications et les plateformes distribuées. Il se base sur les procédures suivantes :

- Les échanges de messages MOM *(Message Oriented Middleware)* où les applications communiquent par échange de messages. Ce modèle se base sur deux principes : le

mode point à point où certains composants produisent les messages et d'autres les consomment ou le mode abonnement (*publish*) où les applications consommatrices de messages s'abonnent à un sujet avant d'utiliser les messages affichés.

- L'appel de procédure à distance RPC *(Remote Procedure Call)* qui représente un protocole réseau permettant de faire des appels de procédures à distance en utilisant les serveurs d'application. Exemple : le web service SOAP *(Simple Object Access Protocol)*, le protocole RPC de Sun de Microsoft, et le RMI *(Remote Method Invocation)* qui représente des composants pour réaliser des appels de procédures à distance.
- La manipulation d'objet à distance, par exemple DCOM technique de manipulation d'objet basée sur le protocole RPC et DCE *(Distributed Computing Environment)* basée sur CORBA.

3. Programmation multitâche et *threads* en C++

Comme nous l'avons dit précédemment, pour parer au manque de performance des technologies classiques, les grands groupes industriels ont opté pour les processeurs multicœurs. En effet, augmenter la puissance des machines en élevant simplement la fréquence de calcul des processeurs monocœurs ne constitue pas la solution idéale, car se pose par la suite le problème de la dissipation de l'énergie thermique générée.

Par conséquent, pour contourner ce problème, tous les grands groupes se sont tournés vers la conception des processeurs multicœurs. Actuellement, il existe plusieurs architectures de processeurs multicœurs. Ce sont des architectures dans lesquelles plusieurs cœurs sont placés dans un même processeur et chacun contient des microprocesseurs indépendants. Comme nous l'avons vu, ces processeurs possèdent plusieurs avantages et sont aussi efficaces dans le traitement multitâche.

Sur un système d'exploitation multitâche, plusieurs processus peuvent être lancés simultanément permettant à celui-ci de passer d'une application à une autre. Par conséquent, un programme multitâche est un programme capable de lancer l'exécution de plusieurs parties de son code simultanément, ce qui signifie que ces parties s'exécutent en parallèle.

Actuellement, avec les développements des processeurs, tous les systèmes d'exploitation (Windows, Linux) sont des systèmes multitâches. La programmation multitâche est aussi bien possible sur les systèmes monoprocesseurs que sur les systèmes multiprocesseurs, mais elle s'avère plus efficace sur ces derniers, car elle permet de repartir les différentes

tâches (exécution en parallèle) sur les processeurs et d'améliorer ainsi les performances des traitements.

La programmation multitâche s'appuie sur l'utilisation des processus et des *threads* systèmes, ce qui permet d'exécuter plusieurs tâches simultanément.

Par exemple, pour gérer les *threads*, Unix utilise l'API POSIX.

Exemple :

- la commande `fork()` : qui crée un processus fils[4];
- `execv()` : qui crée un processus exécutant un programme externe.

3.1 Le principe de fonctionnement des *threads*

Un programme informatique peut être exécuté en plusieurs fonctions qui s'exécutent parallèlement sur un système. Chacune de ces fonctions s'appelle thread.

Un thread est en réalité différent d'un processus. Un processus en exécution peut être composé de plusieurs threads et chacun représente une sous-tâche interne au processus en cours d'exécution. Les *threads* sont souvent appelés processus légers et la création d'un *thread* est dix à cent fois plus rapide que celle d'un processus.

En pratique, lorsqu'on exécute ou invoque un programme, un nouveau processus se crée. À l'intérieur de ce processus, un *thread* se crée, qui exécute le processus. Mais, en même temps, le thread peut créer d'autres *threads* qui vont s'exécuter dans le même processus en prenant simultanément en charge une partie du programme.

La différence entre un thread et un processus peut se résumer ainsi :

- la communication entre les *threads* est plus rapide que celle entre les processus;
- tous les *threads* d'un processus partagent un même espace de mémoire (de travail). Cela rend le partage de l'information facile, mais cette simplicité entraîne des problèmes de synchronisation alors que les processus ont chacun un espace mémoire personnel.

[4] Pour un exemple pas à pas illustrant la fonction `fork()`.

3.2 L'utilisation de POSIX

Avec Linux, il existe plusieurs façons d'implémenter les *threads*. La plus connue est celle qui utilise les *threads* POSIX. Bien que POSIX fut initialement développé pour le langage C, il s'utilise aussi avec le langage C++.

L'API POSIX a pour rôle de « normaliser » l'interfaçage entre les applications et les systèmes d'exploitation. Elle couvre les techniques telles que la programmation réseau par socket, les entrées-sorties sur fichiers, les signaux ou la gestion des *threads*.

Les langages les plus appropriés pour la manipulation de threads sont C et C++. La bibliothèque `pthread` de la norme POSIX contient les primitives pour créer les processus et les synchroniser sous Linux. Ces fonctions ne se trouvent pas dans le langage C ou C++. C'est la raison pour laquelle on fait appel à l'option `-lpthread` lors de l'exécution (édition de liens) des *threads* en ligne de commande.

Lorsque qu'un programme est lancé avec `exec` par exemple, un seul *thread* est créé. Ce *thread* se nomme *thread* initial ou principal.

3.2.1 La création de threads

Comme nous l'avons évoqué précédemment, le fichier entête `pthread.h` contient les primitives et les fonctions de manipulation des *threads*. Ces fonctions permettent entre autres de créer un nouveau *thread*, d'attendre la fin de l'exécution d'un *thread* en cours ou de mettre fin à un *thread*. Dans cette bibliothèque `pthread`, les fonctions les plus importantes sont les suivantes.

`pthread_create ()`

Cette fonction est utilisée pour créer un thread. Elle est similaire à la fonction *fork()* des processus Unix.

La création d'un thread se présente, par exemple, comme suit :

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,
void *(*func) (void *), void *arg);
```

Dans cet exemple nous avons :

- un pointeur sur une variable de type `thread_p` qui va stocker l'identifiant du nouveau *thread*;
- un pointeur vers un objet attribut du *thread* qui va contrôler les détails de l'interaction du *thread* avec le reste du programme. Pour opter pour les valeurs par défaut, on passe alors à `NULL`;

- un pointeur vers la fonction que le *thread* créé doit exécuter. Le *thread* commence par appeler la fonction et se termine soit explicitement par appel à `pthread_exit`, soit implicitement à la fin de la fonction;
- un argument de *thread* de type `void*` qui sera donné à la fonction lors de son exécution.

`pthread_join()`

Cette fonction attend que *le thread* spécifié se termine (équivalent de `waitpid` des processus Unix).

Exemple :

```
int pthread_join (pthread_t tid, void ** status);
```

Si la valeur de `status` est non `NULL` alors la valeur de retour du *thread* est stockée dans l'emplacement pointé par l'argument `status`.

`pthread_exit ()`

Cette fonction représente la fonction pour terminer un thread.

L'exemple suivant montre la création d'un ensemble de *threads* en utilisant la fonction *pthread_create*.

```
#include <stdio.h>
#include <pthread.h>    /*entête de la bibliothèque*/

/* fonction à exécuter par un nouveau thread */
void*
do_loop(void* data)
{
    int i;              /*  compteur,  pour  imprimer  les  numéros
*/
    int j;              /* compteur de délai        */
    int  IDthread  =  *((int*)data);      /*  identifiant  de  thread
*/

    for (i=0; i<10; i++) {
      for (j=0; j<500000; j++) /*  */
          ;
        printf("'%d' - Got '%d'\n", IDthread, i);
    }
```

```
    /* fin du thread*/
    pthread_exit(NULL);
}

Int main(int argc, char* argv[])
{
    int        thr_id;         /* ID pour un nouveau thread */
    pthread_t   p_thread;            /*  structure    du  thread
*/
    int        a         = 1;  /* identifiant du thread 1  */
    int        b         = 2;  /* identifiant du thread 2 */

    /* creation d'un nouveau thread qui execute 'do_loop()' */
    thr_id    =    pthread_create(&p_thread,    NULL,    do_loop,
(void*)&a);
    /* exécuter 'do_loop()' dans le main thread  */
    do_loop((void*)&b);

    /* */
    return 0;
}
```

Le code précédent montre comment créer un ensemble de *threads* en utilisant les fonctions de la bibliothèque `pthread`. Si l'API POSIX est disponible sur votre système, il est bien intéressant d'utiliser les primitives de cette bibliothèque plutôt que les appels système du noyau. Cette bibliothèque offre plusieurs primitives pour gérer les communications et la synchronisation des threads.

Cependant, POSIX offre plus que les primitives. En effet, il permet aussi de gérer les signaux en temps réel, les conflits d'accès en utilisant les sémaphores et les mutex, les verrous sur les emplacements mémoire (variables) et assure une planification priorisée des tâches.

3.2.2 Les mutex

Comme nous l'avons mentionné, tous les *threads* d'un processus partagent la même zone mémoire. Il existe alors des conflits pour accéder à ces zones ou aux variables. Les mutex constituent une solution pour gérer ces conflits.

Un mutex (*mutual exclusion ou* zone d'exclusion mutuelle) est un verrou sur une ressource qui assure un accès unique à cette ressource à un moment donné. Par conséquent, seul le *thread* qui possède le verrou peut manipuler (accéder en lecture ou en écriture) les variables dans la portion de code verrouillé (zone critique). Une fois que le *thread* est terminé, il libère la zone critique et un autre *thread* peut manipuler les ressources.

Une manière simple de créer un mutex est de déclarer une variable de type `pthread_mutex_t` et de l'initialiser avec la constante : `PTHREAD_MUTEX_INITIALIZER`

Exemple :

```
static pthread_mutex_t  mutex = PTHREAD_MUTEX_INITIALIZER;
```

Un mutex créé n'a que deux états possibles. Il est soit verrouillé, soit déverrouillé. On utilise les fonctions `pthread_mutex_lock` et `pthread_mutex_unlock` pour changer les états :

- La fonction `pthread_mutex_lock` peut être utilisée par un *thread* pour verrouiller un mutex. S'il est déverrouillé, il devient verrouillé et sera associé au *thread* appelant la fonction.

  Par contre, s'il est verrouillé par un autre *thread*, alors la fonction `pthread_mutex_lock` suspend le *thread* appelant jusqu'à ce que le mutex soit déverrouillé.

  Exemple :

  ```
  int pthread_mutex_lock (pthread_mutex_t * mutex);
  ```

  Cette fonction essaie de verrouiller un mutex. Le seul argument à lui donner est l'adresse du mutex de type `pthread_mutex_t`.

  En cas de succès, la fonction renvoie 0. En cas d'échec, il renvoie l'une des valeurs suivantes :
  - `EINVAL` : si le mutex n'a pas été initialisé.
  - `EDEADLK` : si le mutex est déjà verrouillé par un *thread* différent.

- La fonction `pthread_mutex_unlock` permet de déverrouiller un mutex.

  Exemple :

  ```
  int pthread_mutex_unlock (pthread_mutex_t * mutex);
  ```

  Comme argument, on lui donne l'adresse du mutex à déverrouiller de type `pthread_mutex_t`.

  En cas de succès la fonction renvoie 0. En cas d'échec elle envoie l'une des valeurs suivantes :

  - `EINVAL` : si le mutex est non initialisé.
  - `EPERM` : si le *thread* appelant ne possède pas le mutex.

4. Introduction à la programmation réseau

Cette partie du texte traite des concepts et des bases de la programmation réseau. La programmation réseau s'effectue par l'intermédiaire des processus, et les sockets constituent un moyen pour faire communiquer les processus entre eux. Les sockets constituent le moyen standard pour la communication sur le réseau. La programmation des sockets s'effectue dans tous les langages de programmation (C, C++, Java etc.)

Souvent on fait l'analogie avec les communications téléphoniques pour décrire une socket. Une socket est considérée comme l'extrémité d'un canal de communication bidirectionnel qui met en liaisons deux applications (processus). Ces deux applications peuvent communiquer par l'envoi de données sur le canal. Cette communication peut se dérouler aussi bien par des processus d'un seul système que par ceux situés sur des systèmes différents sur un réseau. À la différence des communications téléphoniques, les applications des sockets se scindent en deux parties. En effet, Il existe une différence entre les programmes qui attendent les connexions et ceux qui en demandent.

Un serveur est un programme en attente de connexions entrantes sur un port d'écoute et qui propose des services à d'autres programmes.

Un client est un programme qui fait des demandes de services à un serveur et se connecte à celui-ci en utilisant le port d'écoute de connexion. Les services demandés peuvent être l'exécution d'autres programmes, des traitements de données ou bien accéder des ressources stockées.

4.1 Définition

Le terme « socket » désigne l'extrémité d'un canal de communication (point de communication) par lequel un processus peut émettre ou recevoir des données. Il est représenté comme une interface entre les programmes d'application et les protocoles de communication. L'interface de sockets comprend un ensemble de primitives qui permettent gérer l'échange de données entre processus locaux ou distants. Le point de communication est représenté par une variable similaire à un descripteur de fichier.

L'ensemble des primitives sur les sockets permet entre autres :

- d'attribuer aux programmes un rôle de client ou de serveur;
- de rendre transparents les services de communication par échanges de messages;
- de contrôler, gérer et paramétrer les sockets;

- de manipuler les informations de communication entre les processus.

4.2 Les entités et le descripteur d'une socket (revoir cette section)

Comme nous l'avons déjà mentionné, une socket est identifiée par une variable appelée *descripteur*. Ce descripteur qui joue finalement le rôle de nom est retourné par la socket lors de sa création. Ce descripteur ressemble au descripteur d'un fichier, ce qui permet de lui appliquer des primitives telles que `read`, `write`, etc.

Sous Unix, les différentes constantes et fonctions qui sont utilisées pour la gestion des sockets sont définies dans les fichiers `<sys/types.h>` et `<sys/socket.h>`. Il est important d'ajouter ces bibliothèques dans l'en-tête de vos programmes.

Les communications se déroulent entre deux entités, à savoir un client et un serveur. Le dialogue se déroule ainsi.

- Un premier programme crée une socket appelée *serveur* et appelle une méthode bloquante parmi les primitives (`accept()`). Par la suite, le serveur reste en attente d'une demande d'un client.
- Un autre programme ou processus crée une socket appelée client. Le client fournit comme information à sa socket l'adresse IP ainsi que le port d'écoute du serveur.
- Par la suite, les deux sockets seront connectées (dans le cas d'un protocole en mode connecté). Cela a pour effet de débloquer le serveur de son attente. Un canal de flots de données est ainsi mis en place entre les deux entités. À l'aide de ce canal, chaque entité peut lire ou écrire dans sa socket. Les communications par le canal sont bidirectionnelles.
- Une fois le dialogue terminé, chaque entité se déconnecte.

Nous vous présentons maintenant le rôle joué par chaque entité (client et serveur) dans les communications basées sur les sockets.

Le client est un programme qui permet à une machine de se connecter à une autre machine sur laquelle est lancé un serveur.

Lors de son exécution, un programme client réalise les étapes ci-dessous :

- Il doit connaître l'adresse IP et le numéro de port du serveur auquel il doit se connecter.
- Il crée une socket, en mentionnant l'adresse IP et le numéro de port du serveur.
- Il réalise une connexion de sa socket sur la socket du serveur local ou sur celle de la machine distante.

- Il dialogue ensuite avec le programme serveur pour demander des services, faire un traitement ou accéder aux données stockées sur le serveur.
- Il ferme la connexion de sa socket lorsqu'il termine les traitements.

Quant au serveur, c'est un programme qui, une fois lancé sur une machine locale ou ayant accès à un réseau, attend la connexion de clients et leur fournit des informations dont ils ont besoin.

Un programme serveur réalise les étapes suivantes :

- On lui spécifie un numéro de port d'écoute.
- Il crée une socket et la lie sur ce numéro de port.
- Il attend la connexion d'un client.
- Il dialogue (traite des données) avec le client connecté.
- Il ferme la connexion avec le client et se met en attente d'une nouvelle connexion.

4.3 Les domaines d'une socket

Une socket est caractérisée par le domaine de communication où elle se trouve. Ce domaine spécifie les protocoles de communication qui sont utilisés par la socket ainsi que les formats des adresses qui sont utilisés. Deux sockets ne peuvent pas communiquer si elles n'appartiennent pas au même domaine de communication. Il existe plusieurs domaines.

4.3.1 Le domaine Unix

Le domaine Unix (AF_UNIX) permet à deux processus situés sur le même ordinateur de communiquer. Les protocoles de domaine Unix sont une alternative aux communications interprocessus lorsque le client et le serveur sont situés sur la même machine. Cela facilite les communications client-serveur en utilisant la même API qui est utilisée dans le cas où les communications sont distantes.

Les sockets du domaine Unix sont utilisées pour les raisons suivantes :

- La rapidité : les sockets Unix sont deux fois plus rapides qu'une socket TCP lorsqu'elles sont situées sur la même machine.
- Elles sont aussi utilisées lors de la transmission des descripteurs entre des processus sur un même hôte.
- La sécurité : les nouvelles implémentations des sockets de domaine Unix fournissent des informations d'identification du client.

Le domaine Unix utilise le système de fichier comme espace de nom. Ces sockets n'utilisent pas des ports. La structure d'une adresse de socket est définie dans le fichier `<sys/un.h>` :

```
struct sockaddr_un {
   short sun_family ; /* domaine UNIX: AF_UNIX */
   char sun_path[108]; /* référence du fichier */
};
```

`Sun_family` contient la valeur `AF_UNIX` (aussi connue sous le nom `PF_LOCAL`), et `sun_path` contient le chemin d'accès à la socket.

4.3.2 Le domaine Internet

Le domaine Internet (AF_INET) permet les communications externes (IPC externe), c'est-à-dire une communication entre des processus qui sont lancés sur des machines hôtes différentes. Ces communications sont bidirectionnelles à travers le réseau IP et font intervenir une adresse IP et un port d'écoute.

La structure d'une adresse de socket est définie dans le fichier `<netinet/in.h>` :

```
struct in_addr {
  u_long s_addr;
} ;

struct sockaddr_in {
  short sin_family;  // AF_INET représente le domaine internet
  u_short sin_port; //  numéro de port d'écoute
  struct in_addr sin_addr; // l'adresse Internet
  char sin_zero[8]; // un champ de huit zéros
} ;
```

Pour l'adressage IPv6, `AF_INET6` est utilisé à la place de `AF_INET`.

4.4 Les modes de communication

Il existe deux modes de communication sur lesquels se reposent les sockets pour communiquer :

- TCP : c'est le mode connecté *(stream* ou flot*)* utilisant Internet et TCP et le mode connecté utilisant Unix. Ce protocole définit le type de socket `SOCK_STREAM` qui permet une communication bidirectionnelle, sure, séquencée et sans duplication de données.

- UDP : c'est le mode non connecté ou datagramme (*datagram)* utilisant Internet et UDP sur Unix le mode datagramme (*datagram)* avec Unix. Ce protocole définit un type de socket `SOCK_DGRAM` qui permet une communication bidirectionnelle qui est non sure et pas séquencée. Ce qui peut entrainer une duplication des données.

4.4.1 Le mode datagramme

Dans le mode datagramme, les communications et les envois de datagrammes entre les deux sockets se déroulent en mode non connecté. Dans le domaine Internet, le protocole utilisé est UDP, les transferts sont donc non fiables. Et c'est le programme client qui initie la communication en envoyant des messages sans faire une demande de connexion au serveur.

4.4.2 Le mode flot (stream)

Ce mode correspond aux sockets dédiées à la communication en mode connecté. Une demande de connexion est envoyée par le client au serveur qui doit l'accepter avant tout transfert de données. Les communications bidirectionnelles entre le client et le serveur sont séquencées et il n'y a ni perte, ni duplication de données. Ce mode utilise le service TCP et trois phases sont nécessaires lors des communications :

- l'établissement de connexion (demande de connexion et acceptation de connexion);
- le transfert de données;
- la libération de connexion (déconnexion).

Cependant, il faut d'abord avoir créé le socket et lui avoir associé une adresse.

Lors d'une communication entre machines, la première phase consiste à créer la socket.

4.5 Les primitives sur les sockets

Les primitives sont des fonctions qui permettent de manipuler et de gérer les sockets. Ces primitives diffèrent selon le type de sockets. Certaines primitives sont communes aux clients et aux serveurs.

Les primitives sont incluses dans les bibliothèques `types.h` et `socket.h`. Par conséquent, il est nécessaire de porter en en-tête des programmes aussi bien du côté du serveur que du côté des clients les fichiers :

```
#include <sys/types.h>

#include <sys/socket.h>
```

4.5.1 Primitives de création et de suppression de socket

La création d'une socket se fait à l'aide de la primitive socket suivante :

```
int socket(int family, int type, int protocol);
```

Cette primitive retourne `-1` en cas d'erreur de création de la socket ou bien renvoie un entier qui représente le descripteur de la socket en cas de réussite. On manipule la socket créée à l'aide de son descripteur.

Exemple :

```
Nom_descripteur = socket(PF_INET, SOCK_STREAM, 0)
```

Pour fermer la socket, on utilise la fonction *close().*

4.5.2 Primitives pour associer une socket à une adresse

Une fois créée, la socket n'est accessible que par le processus qui initialise sa création. Pour permettre aux autres processus d'utiliser la socket, que les autres processus puissent manipuler la socket, on lui associe une adresse. La primitive `bind` permet de faire cette association entre la socket et une adresse.

Exemple :

```
int bind(int sockfd, const struct sockaddr *addr, int addrlen);
```

Dans cet exemple, la primitive `bind` permet d'affecter l'adresse spécifiée par son pointeur à la socket référencée par le descripteur `sockfd`. Le paramètre `addrlen` représente la longueur de cette adresse et `addr` représente un pointeur vers les informations permettant d'utiliser la socket, à savoir l'adresse locale de la machine ainsi qu'un port d'attache de la socket.

4.5.3 Primitive pour attendre les connexions

Dans le cas du mode connecté, une socket créé du côté du serveur se met en attente de connexion des clients. La primitive `listen` est utilisée par le serveur pour dire qu'il est en

attente de connexion d'un client. Cette primitive prend en argument le descripteur de la socket et le nombre maximum de connexion que le serveur peut accepter.

Exemple :

```
int listen(int sockfd, int maxqueue);
```

Le paramètre `sockfd` représente le descripteur et `maxqueue` représente le nombre maximum de connexions.

4.5.4 Primitive pour accepter une demande de connexion

Pour accepter les connexions des clients, le serveur en attente utilise la primitive `accept` de la socket. Cette primitive est utilisée sur les sockets en mode connecté pour accepter les demandes des connexions des clients. Il faut au préalable utiliser le primitive `listen`.

La primitive `Accept` retourne `-1` en cas d'erreur. Sinon il créé une socket fils avec son descripteur qui prend en charge le client en attente de connexion au serveur.

Exemple :

```
int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

4.5.5 Primitive pour se connecter à une socket

La demande de connexion d'un client à un serveur se fait à l'aide de la primitive `connect`. Elle permet à une socket de se connecter à une socket distant en mettant en place un canal de communication. Il faut spécifier la socket ainsi que le numéro de port de connexion. Il envoie `0` en cas de succès et `- 0` si la connexion échoue.

Exemple :

```
int connect(int sockfd, struct sockaddr *addr, int addrlen);
```

4.5.6 Primitive d'échange de données

Une fois que le canal est mis en place entre le client et le serveur, les communications et les échanges de données peuvent commencer entre les deux sockets. L'émission ou l'écriture des données par une socket se fait à l'aide de la primitive `send` ou `write`. Les données émises par une socket sont reçus par la socket destinataire dans l'ordre de leur émission.

Exemple :

```
int send (int sockfd, char *msg, int msglen, int option)
```

Dans cet exemple, le paramètre `msg` représente l'adresse en mémoire du message à envoyer, `msglen` représente la longueur du message, `sockfd` représente le descripteur de la socket et option qui peut être à `MSG_OOB` dans le cas où le message est prioritaire. Lorsque la primitive `send` est utilisée avec l'option à `0` alors elle se comporte comme une primitive `write`.

La réception des données par une socket se fait à l'aide de la primitive `recv`.

Exemple :

```
int recv (int sockfd, char *msg, int msglen, int option)
```

4.5.7 Fermeture d'une socket

La fermeture d'une socket se fait à l'aide la primitive `shutdown` ou `close`. La primitive `shutdown` prend en paramètre le descripteur de la socket à fermer ainsi qu'un entier qui représente le mode de fermeture. Cet entier possède, la valeur `2` dans le cas où le processus ne désire ni recevoir ni émettre, `0` si le processus ne désire plus recevoir et `1` lorsqu'il ne désire plus recevoir. Par contre, la primitive close prend en argument le descripteur de la socket.

Exemple :

```
int shutdown (int sockfd, 2)
```

4.6 La création d'une socket

Lors d'une communication, la première étape consiste à créer une socket. Cela se fait à l'aide de l'appel-système `socket()`, défini dans `<sys/socket.h>` :

```
int socket (int family, int type, int protocol);
```

En cas d'erreur, la fonction retourne `-1`, sinon elle retourne le descripteur de la socket.

Le premier argument de cette fonction est le domaine (`family`) de communication. Il s'agit d'une constante symbolique pouvant prendre plusieurs valeurs.

famille	***description***
AF_INET	Protocole IPv4
AF_INET6 *AF_UNIX*	Protocole IPv6 Domaine Unix/local
AF_LOCAL	Domaine Unix/local
AF_ROUTE	Socket de routage
AF_KEY	Socket à clé

Nous ne nous intéresserons ici qu'au domaine AF_INET, qui regroupe toutes les communications réseau avec IP, TCP, UDP ou ICMP.

Le deuxième argument (`type`) est le type de socket.

Type	***description***
SOCK_STREAM	Mode connecté avec contrôle de flux
SOCK_DGRAM	Datagramme sans connexion
SOCK_RAW	Dialogue brute

Le troisième argument (`protocol`) indique le protocole désiré et généralement, il suffit d'utiliser la valeur `0`.

Le résultat est une valeur de retour qui représente le descripteur qui est un entier qui identifiera la socket dans tout le reste du programme.

Exemple :

```
int idS ;
idS = socket(AF_INET, SOCK_STREAM, 0) ;
```

Cet exemple crée une nouvelle socket de domaine Internet, de type « mode connecté » et retourne le descripteur dans `idS`.

L'exemple suivant présente les fonctions de création d'une socket en mode connecté.

```
Int cree_socket_stream (const char * nom_hote, const char *
nom_service, const char * nom_proto)
{
int sock;
struct sockaddr_in adresse;
struct hostent * hostent;
struct servent * servent;
struct protoent * protoent;
if ((hostent = gethostbyname(nom_hote)) == NULL) {
perror("gethostbyname");
return -1;
}
if ((protoent = getprotobyname(nom_proto)) == NULL) {
perror("getprotobyname");
return -1;
}
if ((servent = getservbyname(nom_service,
protoent->p_name)) == NULL) {
perror("getservbyname");
return -1;
}
if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("socket");
return -1;
}
memset(& adresse, 0, sizeof (struct sockaddr_in));
adresse sin_family = AF_INET;
adresse.sin_port = servent->s_port;
adresse.sin_addr . s_addr =
((struct in_addr *) (hostent->h_addr))->s_addr;
if (bind(sock, (struct sockaddr *) & adresse,
sizeof(struct sockaddr_in)) < 0) {
```

```
close(sock);
perror("bind");
return -1;
}
return sock;
}
```

4.7 Communication

Pour attacher une socket à une adresse on utilise la primitive `bind`.

La syntaxe est la suivante :

```
int bind(sock, localaddr, addrlen),
int sock, /*descripteur de socket*/
struct sockaddr_in *localaddr, /*adresse de socket
                               /*locale*/
int addrlen ; /*longueur de l'adresse*/
```

Le résultat est un nombre négatif si l'opération échoue.

Remarque : avant d'appeler la primitive `bind`, il est nécessaire d'initialiser les champs de la structure de type `sockaddr_in` à zéro en utilisant la primitive `bzero()` et de remplir les valeurs correctes pour les champs ci-dessous :

- le domaine de communication ;
- le numéro de port (soit fixé, soit 0) ;
- l'adresse IP de la machine.

La syntaxe de la primitive `bzero()` qui remet à zéro les `nboctets` à partir de l'adresse est :

```
addr_struct : bzero(addr_struct, nboctets),
char *addr_struct, /*adresse de la socket locale*/
int nboctets; /*taille de la structure*/
```

4.8 Les fonctions usuelles sur les sockets

Lors de la programmation socket, il est nécessaire de disposer d'un certain nombre de fonctions. Il s'agit de :

- La fonction `gethostbyname` qui retourne les caractéristiques d'une machine.
- La fonction `gethostbyaddr` qui récupère les caractéristiques d'une machine.

- La fonction `gethostent` qui récupère les caractéristiques d'une machine.
- La fonction `memset` qui initialise une zone mémoire.
- La fonction `memcpy` qui recopie une zone mémoire.
- La fonction `htonl(3N)` : conversion d'un entier long
- *La fonction* `htons` qui convertit un entier court.
- *La fonction* `ntohl` qui convertit un entier long.
- *La fonction* `ntohs` qui convertit un entier court.

4.8.1 Création d'une socket côté client

Voici un exemple de création et manipulation d'une socket côté client. La socket représente un client qui fait une demande de connexion à un serveur. Le client reçoit l'adresse IP et le port de connexion du serveur. Une fois la connexion établie, le client envoie au serveur un message et reste en attente de la réponse du serveur.

```
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>
#define MAX_LINE 100

#define LINE_ARRAY_SIZE (MAX_LINE+3)
using namespace std;
int main()
{
int socketDescriptor;
int numRead;
unsigned short int serverPort;
struct sockaddr_in serverAddress;
struct hostent *hostInfo;
struct timeval timeVal;
fd_set readSet;
char buf[LINE_ARRAY_SIZE], c;

cout << "Donnez le nom du serveur ou son adresse IP: ";
memset(buf, 0x0, LINE_ARRAY_SIZE);  // Mettre à zero le tampon
cin.get(buf, MAX_LINE, '\n');

//gethostbyname() reçoit un nom d'hôte ou une adresse IP sur 4
//octets et renvoie un pointeur sur une structure hostent.
hostInfo = gethostbyname(buf);
if (hostInfo == NULL) {
cout << "interprétation du problème " << buf << "\n";
exit(1);
}
cout << "entrer le numéro de port du serveur: ";
```

```
cin >> serverPort;
cin.ignore(1,'\n'); // suppression du saut de ligne
// Création de socket de domaine internet "AF_INET" c'est-à-dire
// IPv4 et de type datagramme "SOCK_DGRAM" en utilisant le
protocole // UDP la valeur 0 indique qu'un seul protocole sera
utilisé avec  // cette socket.
socketDescriptor = socket(AF_INET, SOCK_DGRAM, 0);
if (socketDescriptor < 0)
           {
cerr << "cannot create socket\n";
exit(1);
            }
// Initialisation des champs de  serverAddress
serverAddress.sin_family = hostInfo->h_addrtype;
memcpy((char *) &serverAddress.sin_addr.s_addr,
hostInfo->h_addr_list[0], hostInfo->h_length);
serverAddress.sin_port = htons(serverPort);

cout << "\nEntrez quelques caractères au clavier.\n";
cout << "Le serveur les modifiera et les renverra.\n";
cout << "Pour sortir, entrez une ligne avec le caractère '.'
uniquement\n";
cout << "Si une ligne dépasse " << MAX_LINE << " caractères,\n";
cout << "seuls les " << MAX_LINE << " premiers caractères seront
utilisés.\n\n";

//  Invite  de  commande  pour  l'utilisateur  et  lecture  des
caractères // jusqu'à la limite MAX_LINE. Puis suppression du
saut de ligne.
cout << "Input: ";
memset(buf, 0x0, LINE_ARRAY_SIZE);
cin.get(buf, MAX_LINE, '\n');

//  suppression  des  caractères  supplémentaires  et  du  saut  de
ligne
cin.ignore(1000,'\n');

//  Arrêt  lorsque  l'utilisateur  saisit  une  ligne  ne  contenant
qu'un // point
while (strcmp(buf, "."))
                      {
// Envoi de la ligne saisie au serveur
if (sendto(socketDescriptor, buf, strlen(buf), 0,
(struct sockaddr *) &serverAddress,
sizeof(serverAddress)) < 0)
 {
cerr << "cannot send data ";
close(socketDescriptor);
```

```
exit(1);
                }

// Attente de la réponse pendant une seconde.
FD_ZERO(&readSet);
FD_SET(socketDescriptor, &readSet);
timeVal.tv_sec = 1;
timeVal.tv_usec = 0;

if (select(socketDescriptor+1, &readSet, NULL, NULL, &timeVal))
           {
// Lecture de la ligne modifiée par le serveur.
memset(buf, 0x0, LINE_ARRAY_SIZE);
numRead = recv(socketDescriptor, buf, MAX_LINE, 0);
if (numRead < 0)
{
  cerr << "pas de reponse du serveur ?";
  close(socketDescriptor);
  exit(1);
}
cout << "Modified: " << buf << "\n";
     }
      else {
      cout << "—pas de réponse du serveur  dans 1 seconde.\n";
}
//  Invite  de  commande  pour  l'utilisateur  et  lecture  des
caractères // jusqu'à la limite MAX_LINE. Puis suppression du
saut de ligne.
// Comme ci-dessus.
cout << "Input: ";
memset(buf, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon
cin.get(buf, MAX_LINE, '\n');
//  Suppression  des  caractères  supplémentaires  et  du  saut  de
ligne
cin.ignore(1000,'\n');
}
close(socketDescriptor);    // Fermeture de la socket
return 0;
}
```

4.8.2 Création d'une socket côté serveur

Du côté du serveur, il faut aussi créer une socket, ce qui permet de mettre en place un canal de communication entre ce serveur et les clients. On crée la socket avec la primitive socket :

```
int listenSocket;
```

```
<snipped/>
listenSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (listenSocket < 0) {
cerr << "cannot create listen socket";
exit(1);
}
```

Cette étape est similaire à celle du côté client : on relie le numéro de la socket avec le numéro de port d'écoute choisi.

```
int listenSocket;
struct sockaddr_in serverAddress;
<snipped/>
serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(listenPort);
if (bind(listenSocket•,
(struct sockaddr *) &serverAddress, sizeof(serverAddress)¸) < 0)
{
  cerr << "Échec du bind socket";
  exit(1);
}
```

On utilise l'argument `INADDR_ANY` pour spécifier que le programme est en écoute sur toutes les adresses IP sources. `listenSocket` contient le résultat de l'appel de la fonction socket (i.e. le numéro du canal de communication). `(struct sockaddr *) &serverAddress` et `sizeof(serverAddress)` représentent la structure de désignation de l'adresse IP et du numéro de port du serveur, suivi de la taille de cette structure.

Apres l'appel de `bind`, le programme reste en attente (`listen`) des données du client.

```
int listenSocket;
struct sockaddr_in clientAddress;
char line[(MAX_MSG+1)];
<snipped/>
listen(listenSocket, 3);
<snipped/>
if (recvfrom(listenSocket, line, MAX_MSG•, 0, (struct sockaddr
*) &clientAddress,
&clientAddressLength) < 0)
{
  cerr << " I/O Problem";
  exit(1);
}
```

La primitive `listen` spécifie que le serveur écoute sur le canal de communication. Le paramètre `3` spécifie le nombre maximum de connexions clients que le serveur supporte.

Les paramètres `line` et `MAX_MSG` correspondent au datagramme et à sa longueur. Enfin, la primitive `sendto` est utilisée pour envoyer les données d'une socket à une autre.

Le programme complet est présenté ci-dessous :

```
#include <arpa/inet.h>
#include <netdb.h>
#include <netinet/in.h>
#include <unistd.h>
#include <iostream>
#define MAX_LINE 100

#define LINE_ARRAY_SIZE (MAX_LINE+3)
using namespace std;
int main()
{
int listenSocket, i;
unsigned short int listenPort;
socklen_t clientAddressLength;
struct sockaddr_in clientAddress, serverAddress;
char line[LINE_ARRAY_SIZE];
cout << " donnez le numéro d'écoute (entre 1500 et 65000): ";
cin >> listenPort;

/// Création de socket en écoute et attente des requêtes des
clients
listenSocket = socket(AF_INET, SOCK_DGRAM, 0);
if (listenSocket < 0) {
cerr << "cannot create listen socket";
exit(1);
}

// /Connexion du socket au port en écoute.
// /On commence par initialiser les champs de la structure
serverAddress puis
// /on appelle bind(). Les fonctions htonl() et htons()
convertissent respectivement les entiers
 /// longs et les entiers courts du rangement hôte

serverAddress.sin_family = AF_INET;
serverAddress.sin_addr.s_addr = htonl(INADDR_ANY);
serverAddress.sin_port = htons(listenPort);
if (bind(listenSocket,
(struct sockaddr *) &serverAddress,
sizeof(serverAddress)) < 0) {
cerr << "cannot bind socket";
exit(1);
}
```

```
// Attente des requêtes clients.
// C'est un appel non-bloquant ; c'est-à-dire qu'il enregistre
ce programme
// auprès du système comme devant attendre des connexions sur
cette socket avec
// cette tâche. Puis, l'exécution se poursuit.
listen(listenSocket, 5);
cout << "Waiting for request on port " << listenPort << "\n";
while (1) {

clientAddressLength = sizeof(clientAddress);
// Mise à zéro du tampon de façon à connaître le délimiteur
// de fin de chaîne.
memset(line, 0x0, LINE_ARRAY_SIZE);
if (recvfrom(listenSocket, line, LINE_ARRAY_SIZE, 0,
(struct sockaddr *) &clientAddress,
&clientAddressLength) < 0) {
cerr << " I/O Problem";
exit(1);
}

// Affichage de l'adresse IP du client.
cout << " de " << inet_ntoa(clientAddress.sin_addr);
// Affichage du numéro de port du client.
cout << ":" << ntohs(clientAddress.sin_port) << "\n";
// Affichage de la ligne reçue
cout << " Reçu: " << line << "\n";
// Conversion de cette ligne en majuscules.
for (i = 0; line[i] != '\0'; i++)
line[i] = toupper(line[i]);
// Renvoi de la ligne convertie au client.
if (sendto(listenSocket, line, strlen(line) + 1, 0,
(struct sockaddr *) &clientAddress,
sizeof(clientAddress)) < 0)
cerr << "Error: Ne peut envoyés ces données";
memset(line, 0x0, LINE_ARRAY_SIZE); // Mise à zéro du tampon
}
}
```

5. Bibliothèques scientifiques en C++

C++ regroupe plusieurs bibliothèques qui ont été fusionnées et normalisées pour permettre la portabilité des programmes. Ces bibliothèques sont constituées de classes et de fonctions standardisées. Ces classes et fonctions sont développées par plusieurs

fournisseurs d'environnements et unifiées pour donner des bibliothèques standard pour ce langage. Une des librairies les plus utilisées est STL (*Standard Template Library*), vue dans le précédent module.

L'objectif de ces bibliothèques est de donner des fonctions générales réutilisables par les programmeurs. Ces derniers n'ont pas à reprogrammer ces fonctions mais peuvent simplement les utiliser.

Les fonctions des bibliothèques sont utilisées comme toute autre fonction dans un programme à la différence qu'on les utilise par simple appel sans se préoccuper de leur code source (ou implémentation) qui, lui, est incorporé dans les bibliothèques et non dans le code source du programme. La directive de compilation `#include` permet d'introduire le prototype de la fonction de bibliothèque à votre programme.

Cette section 5 a pour but de présenter les principales fonctionnalités de la bibliothèque standard C++ et de tirer profit de certaines de ces bibliothèques pour la programmation mathématique.

L'utilisation des librairies développées avec des algorithmes mathématiques a largement démontré leur fiabilité et leur niveau d'optimisation, tant au niveau de l'utilisation de la mémoire qu'au niveau des performances.

En C++, il existe plusieurs bibliothèques de fonctions. Nous avons déjà vu quelques librairies au cours des modules précédents. Ici, nous allons décrire quelques librairies intéressantes pour la programmation scientifique.

5.1 Les bibliothèques scientifiques

5.1.1 GSL (GNU Scientific Library)

GSL ou *GNU Scientific Library* est une librairie numérique *open source* destinée aux programmeurs de C et de C++. Elle offre de nombreux outils ainsi que plusieurs fonctions de manipulation de données scientifiques. Elle est riche, simple d'utilisation, facile à compiler et ne requiert pas de dépendances avec d'autres paquets. Parmi les domaines couverts par les fonctions de la bibliothèque, mentionnons les calculs de permutation des nombres, les calculs statistiques, l'algèbre linéaire, les équations différentielles.

5.1.2 GMP (GNU Multiple Precision arithmetic library)

GMP est une bibliothèque portable écrite en C pour les calculs arithmétiques en multiprécision. Elle est *open source* et fait partie du projet GNU. Les domaines dans lesquels on peut l'utiliser sont, entre autres, la cryptographie, les calculs algébriques, la sécurité des applications internet. Elle incorpore plusieurs fonctions et des classes utilisables aussi bien en C qu'en C++.

5.1.3 Boost

Boost offre un ensemble d'outils écrits en C++ par une communauté de développeurs. C'est un ensemble de bibliothèques pour développer des codes de calcul. Boost existe aussi bien en version Linux qu'en version Windows.

Certaines librairies de Boost sont actuellement incluses dans le nouveau standard C++11. Parmi les librairies, mentionnons entre autres :

- Asio, thread, MPI pour la programmation concurrente.
- Geometry pour les calculs géométriques.
- Graph pour les graphes.
- Multidimensionnal Array pour manipuler les tableaux à plusieurs dimensions.
- Pool, Smart Ptr, Utility pour la gestion de la mémoire.
- Python pour interfacer avec python.
- Etc.

5.1.4 GLM (OpenGL Mathematics)

GLM ou *OpenGL Mathematics* est une bibliothèque C++ libre de OpenGL qui comprend un ensemble de classes et de fonctions utilisées pour les applications en 3D et les jeux basés sur les spécifications du langage GLSL (*OpenGL Shading Language)*. Sa syntaxe est très proche de celle de GLSL, ce qui facilite les développements pour les programmeurs qui connaissent déjà le langage GLSL.

5.1.5 LAPACK++

LAPACK++ est une librairie pour l'algèbre linéaire numérique. Elle supporte les plateformes Linux et Windows. Contrairement à LAPACK *(Linear Algebra PACKage)* qui est écrit en Fortran, LAPACK++ est écrit en C++.

Elle intègre les classes comme :

- *LaGenMatDouble* (réel) et *LaGenMatComplex* (complexe) pour manipuler les matrices.
- *LaVectorDouble* et *LaVectorComplex* pour manipuler les vecteurs.

5.2 La bibliothèque mathématique

La bibliothèque `Math` fournit un ensemble de fonctions pour effectuer des opérations mathématiques. C'est la librairie la plus utilisée pour les calculs mathématiques en C++.

Voici une partie des fonctions de la bibliothèque `math.h`.

`cos`	Donne le cosinus d'un angle (radian)
`acos`	Calcule l'arc-cosinus d'un nombre
`cosh`	Calcule le cosinus hyperbolique
`sin`	Calcule le sinus d'un angle
`asin`	Calcule l'arc sinus
`sinh`	Calcule le sinus hyperbolique
`tan`	Calcule la tangente de l'angle
`atan`	Calcule l'arc tangente
`tanh`	Calcule la tangente hyperbolique
`atan2`	Calcule l'arc tangente
`exp`	L'exponentielle d'un nombre
`ldexp`	multiplie un nombre par une puissance entière de 2 `ldexp(x,n) = x . 2n`
`log`	Logarithme népérien
`log 10`	Logarithme à base 10
`pow`	Puissance d'un nombre
`sqrt`	Racine carré d'un nombre
`cbrt`	La racine cubique du nombre
`modf`	Renvoie la partie entière du nombre et stocke la partie décimale dans

`ptr (double modf(double x, double *ptr))`

`fmod` Reste réel de la division de `x` par `y`
`double fmod(doublex, doubley)`

`floor` Renvoie la partie entière inférieure d'un nombre

`ceil` Renvoie la partie entière supérieure d'un nombre

`fabs` Valeur absolue du nombre

L'exemple suivant présente une fonction qui calcule la surface d'un cercle dont le rayon est donné par l'utilisateur. Ce programme utilise la fonction `pow` pour calculer la surface du cercle.

```
#include <iostream>
#include <cmath>
using namespace std;
int main(void)
{
   double rayon, surface;
   cout << "saisissez le rayon du cercle: ";
   cin >> rayon;
   surface = 3.14159 * pow(rayon, 2);
   cout << "la surface du cercle est " << surface << endl;
   return 0;
}
```

Résultat :

```
Entrer le rayon du cercle : 6
La surface est 113.097
```

L'exemple suivant vérifie l'identité de l'équation `sin 2x = 2 sin x cos x`

```
int main()
{
  // test l'identité de sin 2x = 2 sin x cos x:
  for (float x=0; x < 2; x += 0.2)
  cout << x << "\t\t" << sin(2*x) << "\t" << 2*sin(x)*cos(x) <<
endl;
}
```

Le résultat du programme imprime `x` sur la première colonne, puis `sin 2x` sur la deuxième colonne et `2sin x cos x` sur la troisième colonne :

```
0                       0                                   0
```

```
0.2                     0.389418                0.389418
0.4                     0.717356                0.717356
0.6                     0.932039                0.932039
0.8                     0.999574                0.999574
1                       0.909297                0.909297
1.2                     0.675463                0.675463
1.4                     0.334988                0.334988
1.6                     -0.0583744              -0.0583744
1.8                     -0.442521               -0.442521
```

5.3 Les patrons de classes

La bibliothèque standard offre quelques patrons de classes pour faciliter les opérations mathématiques usuelles, telles que la manipulation des nombres complexes et les vecteurs, de manière à doter C++ de possibilités voisines de celles de Fortran 90 et à favoriser son utilisation sur des calculateurs vectoriels ou parallèles. Il s'agit essentiellement :

- des classes `complex`;
- des classes `valarray` et des classes associées.

Également, on dispose de classes `bitset` qui permettent de manipuler efficacement des suites de bits.

5.3.1 La classe `complex`

Le patron de classe `complex` offre de très riches outils de manipulation des nombres complexes. Il peut être paramétré par n'importe quel type `float`, `double` *ou* `long double`.
Il comporte :

- les opérations arithmétiques usuelles : `+`,`–`, `*`, `/`
- l'affectation (ordinaire ou composée : `+=`, `–=`, etc.)
- les fonctions de base :
  - `abs` : module
  - `arg` : argument
  - `real` : partie réelle
  - `imag` : partie imaginaire
  - `conj` : complexe conjugué
- les fonctions « "transcendantes »" :
  - `cos, sin, tan`
  - `acos, asin, atan`
  - `cosh, sinh, tanh`
  - `exp, log`

- le patron de fonctions `polar` (paramétré par un type) qui permet de construire un nombre complexe à partir de son module et de son argument.

Voici un exemple d'utilisation de la plupart de ces possibilités.

```
#include <iostream>
#include <complex>
using namespace std ;
main()
{     complex<double> z1(1, 2), z2(2, 5), z, zr ;
cout << "z1 : " << z1 << " z2 : " << z2 << "\n" ;
cout << "Re(z1) : " << real(z1) << " Im(z1) : " << imag(z1)
<< "\n" ;
cout << "abs(z1) : " << abs(z1) << " arg(z1) : " << arg(z1)
<< "\n" ;
cout << "conj(z1) : " << conj(z1) << "\n" ;
cout << "z1 + z2 : " << (z1+z2) << " z1*z2 : " << (z1*z2)
<< " z1/z2 : " << (z1/z2) << "\n" ;
complex<double> i(0, 1) ; // on definit la constante i
z = 1.0+i ;
zr = exp(z) ;
cout << "exp(1+i) : " << zr << " exp(i) : " << exp(i) <<
"\n" ;
zr = log(i) ;
cout << "log(i) : " << zr << "\n" ;
zr = log(1.0+i) ;
cout << "log(1+i) : " << zr << "\n" ;

double rho, theta, norme ;
rho = abs(z) ; theta = arg(z) ; norme = norm(z) ;
cout << "abs(1+i) : " << rho << " arg(1+i) : " << theta
<< " norm(1+i) : " << norme << "\n" ;
double pi = 3.1415926535 ;
cout << "cos(i) : " << cos(i) << " sinh(pi*i): " <<
sinh(pi*i)
<< " cosh(pi*i) : " << cosh(pi*i) << "\n" ;
z = polar<double> (1, pi/4) ;
cout << "polar (1, pi/4) : " << z << "\n" ;
}
```

Résultat :

```
z1 : (1,2) z2 : (2,5)
Re(z1) : 1 Im(z1) : 2
abs(z1) : 2.23607 arg(z1) : 1.10715
conj(z1) : (1,-2)
z1 + z2 : (3,7) z1*z2 : (-8,9) z1/z2 : (0.413793,-0.0344828)
```

```
exp(1+i) : (1.46869,2.28736) exp(i) : (0.540302,0.841471)
log(i) : (0,1.5708)
log(1+i) : (0.346574,0.785398)
abs(1+i) : 1.41421 arg(1+i) : 0.785398 norm(1+i) : 2
cos(i) : (1.54308,0) sinh(pi*i): (0,8.97932e-011) cosh(pi*i) :
(-1,0)
polar (1, pi/4) : (0.707107,0.707107)
```

5.3.2 La classe `valarray` *et les classes associées*

Le patron de classes `valarray` est adapté à la manipulation de vecteurs (mathématiques), c'est-à-dire de tableaux numériques. Il offre des possibilités de calcul vectoriel comparables à celles qu'on trouve dans un langage de programmation scientifique comme Fortran 90. En outre, quelques classes utilitaires permettent de manipuler des sections de vecteurs; certaines d'entre elles facilitent la manipulation de tableaux à plusieurs dimensions (deux ou plus).

On peut construire des vecteurs dont les éléments sont d'un type de base comme `bool`, `char`, `int`, `float`, `double` ou d'un type `complex`.

Voici quelques exemples de construction.

```
#include <valarray>
using namespace std ;
.....
valarray<int> vi1 (10) ;      /*   vecteur   de   10   int   non
initialisés */
valarray<float>  vf  (0.1,  15)  ;  /*  vecteur  de  15  float
initialisés à 0.1 */
int t[] = {1, 3, 5, 7} ;
valarray <int> vi2 (t, 4) ;   /* vecteur de 4 int intialisé avec
les */
/* 4 (premières) valeurs de t */
valarray <complex<float> > vcf (15) ; /* vecteur de 15 complexes
*/
valarray <int> v ;            /* vecteur vide pour l'instant */
```

5.3.3 L'opérateur `[]`

Une fois un vecteur construit, on peut accéder à ses éléments de façon classique en utilisant l'opérateur `[]` (comme pour l'accès à une cellule d'un tableau) comme dans :

```
valarray <int> vi (4) ; int n, i ;
.....
v[3] = 1 ;
n = v[i] + 2 ;
```

Aucune protection n'est prévue sur la valeur utilisée comme indice.

5.3.4 L'affectation et le changement de taille

Il est possible de faire des affectations entre les vecteurs dont les éléments sont de même type, même s'ils n'ont pas la même dimension, c'est-à-dire le même nombre d'éléments. Par contre, on ne peut pas faire cette affectation si les éléments sont de types différents.

```
valarray <float> vf1 (0.1, 15) ; /* vecteur de 15 float égaux à
0.1 */
valarray <float> vf2 (0.5, 10) ; /* vecteur de 10 float égaux à
0.5 */
valarray <float> vf3 ;           /* vecteur vide pour l'instant
*/
valarray <int> vi (1, 10) ;      /* vecteur de 10 int égaux à 1
*/
.....
vf1 = vf2 ;   * OK vf1 et vf2 sont deux vecteurs de 10 float
égaux à 0.5 */
/* les anciennes valeurs de vf1 sont perdues */
vf3 = vf2 ;       /* OK ; vf3 comporte maintenant 10 éléments */
vf1 = vi ;        /* incorrect vu que vf1 et vi sont de type
différent */
/* mais on peut faire : */

/* for (i=0 ; i<vf1.size() ; i++) vf1[i] = vi [i] ; */
```

5.3.5 La fonction resize

Cette fonction permet de modifier la taille d'un vecteur :

```
vf1.resize(10) ;  /* vf1 comporte maintenant 10 éléments : les
10 */
/* premiers ont conservé leur valeur */
vf3.resize (6) ;  /* vf3 ne comporte plus que 6 éléments (leur
*/
/* valeur n'a pas changé)
```

5.3.6 Le calcul vectoriel

Les classes `valarray` permettent d'effectuer des opérations de calcul vectoriel en généralisant le rôle des opérateurs et des fonctions numériques.

Un opérateur unaire (opérateur sur un seul vecteur) appliqué à un vecteur fournit en résultat un vecteur obtenu en appliquant cet opérateur à chacun de ses éléments.

Exemple :

```
valarray<float> v1(5), v2(5), v3(5) ;
.....
v3 = -v1 ;                    /* v3[i] = -v1[i] pour i de 0 à 4
*/
v3 = cos(v1) ;                /* v3[i] = cos(v1[i]) pour i de 0 à
4 */
```

Un opérateur binaire appliqué à deux vecteurs de même taille fournit en résultat le vecteur obtenu en appliquant cet opérateur à chacun des éléments de même rang.

Exemple :

```
valarray<float> v1(5), v2(5), v3(5) ;
.....
v3 = v1 + v2 ;           /* v3[i] = v2[i] + v1[i] pour i de 0 à
4 */
v3 = v1*v2 + exp(v1) ;   /*  v3[i]  =  v1[i]*v2[i]  +  exp(v1[i])
pour i de /* 0 à 4 */
```

De la même manière, il est possible d'appliquer une fonction de son choix à tous les éléments d'un vecteur en utilisant la fonction membre `apply`. Par exemple, si `fct` est une fonction recevant un `float` et retournant un `float` :

```
v3 = v1.apply(fct) ; /* v3[i] = fct (v1[i]) pour i de 0 à 4 */
```

Également, il existe les opérateurs de comparaison (`==`, `!=`, `<`, `<=`) qui s'appliquent à deux opérandes (de type `valarray`) ayant le même nombre d'éléments et qui fournissent en résultat un vecteur de booléens.

Exemple :

```
int dim = ... ;
valarray<float> v1(dim), v2(dim) ;
valarray<bool> egal(dim), inf(dim) ;
.....
egal = (v1 == v2) ;     /* egal[i] = (v1[i] == v2[i]) pour i de
0 à dim-1 */
inf = (v1 < v2) ;       /* inf[i] = (v1[i] < v2[i]) pour i de 0
à dim-1 */
```

5.3.7 Les fonctions de comparaison

`max` et `min` permettent d'obtenir le plus grand ou le plus petit élément d'un vecteur.

```
int vmax, vmin, t[] = { 2, 10, 20, 4, 7, 6} ;
valarray <int> vi (t, 6) ;
.....
vmax = max (vi) ;             /* donne vmax = 20 */
vmin = min (vi) ;         /* donne vmin = 2 */
```

5.3.8 La sélection de valeurs par masque

En utilisant un masque, il est possible de sélectionner certaines valeurs d'un vecteur afin de former un nouveau vecteur de taille inférieure ou égale. Un masque est un vecteur de type `valarray<bool>` dans lequel chacun des éléments précise si l'on sélectionne (`true`) ou non (`false`) l'élément correspondant.

Exemple :
Supposons que l'on dispose de ces déclarations :

```
valarray <bool> masque(6) ;         /*  on  suppose  que  masque
contient : */
/* true true false true false false */
valarray <int> vi(6) ;              /* on suppose que vi contient
: */
/* 5 8 2 7 3 15 */
```

L'expression `vi[masque]` désigne un vecteur formé des seuls éléments de `vi` pour lesquels l'élément correspondant de masque a la valeur `true`. Ce qui donne un vecteur de trois entiers `(5,8,7)`.

Par exemple :

```
valarray<int> vi1 ; /* vecteur vide */
.....
vi1 = vi[masque] ; /* vi1 est un vecteur de 4 entiers : 5, 8, 7,
9 */
```

Les affectations peuvent se faire des deux côtés. En voici deux exemples utilisant les mêmes déclarations que ci-dessus :

```
vi[masque] = -1 ;/* place la valeur -1 dans les éléments de vi
pour */
/* lesquels la valeur de l'élément correspondant */
/* de masque est true */

valarray<int> v(12, 5) ; /* vecteur de 5 éléments */
```

```
vi[masque] = v ;  /* recopie les premiers éléments de v dans les
*/
/* éléments de vi pour lesquels la valeur de */
/* l'élément correspondant de masque est true */
```

Exemple de programme complet illustrant ces différentes possibilités :

```
#include <iostream>
#include <valarray>
#include <iostream>
#include <iomanip>
#include <valarray>
using namespace std ;
main()
{ int i ;
        int t[] = { 1, 2, 3, 4, 5, 6} ;
        bool mt[] = { true, true, false, true, false, true} ;
  valarray <int> v1 (t, 6), v2 ;               // v2 est vide
        valarray <bool> masque (mt, 6) ;
        v2 = v1[masque] ;
        cout << "v2 : " ;
        for (i=0 ; i<v2.size() ; i++) cout << setw(4) << v2[i] ;
        // setw(n) Spécifie que la prochaine sortie s'effectuera
sur n
        // caractères
        cout << "\n" ;

        v1[masque] = -1 ;
        cout << "v1 : " ;
        for (i=0 ; i<v1.size() ; i++) cout << setw(4) << v1[i] ;
        cout << "\n" ;

  valarray <int> v3(8); /* il faut au moins 4 elements dans v3
*/
        for (i=0 ; i<v3.size() ; i++) v3[i] = 10*(i+1) ;
        v1[masque] = v3 ;
        cout << "v1 : " ;
        for (i=0 ; i<v1.size() ; i++) cout << setw(4) << v1[i] ;
        cout << "\n" ;
}
```

Résultat de l'exécution :

```
v2 : 1 2 4 6
v1 : -1 -1 3 -1 5 -1
v1 : 10 20 3 30 5 40
```