UNIVERSITÉ DE
SHERBROOKE

Département d'informatique
Faculté des sciences

IFT 630 - Processus concurrents et parallélisme

Chapitre 5

Communication inter-processus

Gabriel Girard[1]

Sherbrooke

26 janvier 2023

[1] Gabriel.Girard@usherooke.ca

Table des matières

Chapitre 5

Communication inter-processus

Ce chapitre est inspiré de [23, 34, 35].

La communication inter-processus est un concept par lequel un processus a la capacité d'influencer l'exécution d'un autre processus. Elle peut se faire de deux façons :

- la mémoire partagée ;

  Cette technique requiert que les processus partagent des ressources (ou des variables). Les processus échangent ensuite de l'information par l'intermédiaire de ces ressources. Le concept des producteurs/consommateurs déjà présenté en est un exemple, car ceux-ci communiquent via un tampon partagé.

  Élaborer les mécanismes de cette communication est à la charge de la personne responsable de la programmation. Celle-ci doit également spécifier la synchronisation. Le système, lui, ne fournit que la possibilité de partager les ressources et les outils de synchronisation.

- le passage de messages.

  Avec la technique dite «passage de messages», c'est le système qui pourvoit seul au transfert d'informations en procurant tous les outils nécessaires et en assurant automatiquement la synchronisation.

Remarquons que les deux techniques peuvent co-exister à l'intérieur des systèmes sous-jacents. Toutefois, ce chapitre est exclusivement consacré à la communication par passage de messages. Dans un système «purement» orienté vers la transmission de messages, les processus ne partagent pas de mémoire (ou presque). La communication se fait sans aucune variable partagée, du moins de façon visible par la personne qui développe le programme.

Un système de communication inter-processus basé sur le passage par messages fournit plusieurs primitives, le plus souvent implantées par des appels systèmes. Les deux principales primitives sont :

- `send` *msg* `to` *dest*

  Cette primitive permet d'envoyer un message «*msg*» à un processus identifié par «*dest*».

- `receive` *msg* `from` *source*

Cette primitive permet de recevoir un message, qui sera emmagasiné dans la ressource «*msg*», en provenance d'un processus identifié par «*source*».

Pour enrichir ces fonctionnalités de base, certains environnements procurent d'autres primitives :

- `wait`

  Cette primitive permet d'attendre un événement relié à l'envoi ou à la réception de message.

- `sendReply`

  Cette primitive permet d'envoyer une réponse à un message déjà reçu. Elle fait généralement suite à une réception de message.

- `receiveReply`

  Cette primitive permet de recevoir une réponse à un message déjà envoyé (une demande de service par exemple). Elle fait généralement suite à un envoi de message.

L'avantage principal de la communication par passage de messages est de ne nécessiter aucune synchronisation explicite. En effet, un message ne peut être reçu qu'après son envoi ! Généralement la réception est donc bloquante puisqu'il y a attente de l'arrivée d'un message. La synchronisation lors de l'envoi d'un message s'implante de diverses façons dont nous traiterons plus loin.

Lorsque l'on conçoit les primitives de communication, on doit choisir leur forme et leur sémantique. Deux problèmes se posent :

- Comment spécifier les noms de la source et de la destination ?
- Comment synchroniser la communication ?

5.1 Canal de communication

Différentes façons de faire sont possibles pour spécifier la source et la destination d'un message. De façon générale, leur désignation sert à établir un lien logique appelé **canal de communication**. Un canal est représenté par un couple

«**(source,destination)**»

Il existe aussi plusieurs techniques pour spécifier le canal, appelées techniques de désignation :

- la désignation directe
- le port global (boîte aux lettre)
- le port

5.1.1 La désignation directe

La **désignation directe** est de loin la plus simple pour spécifier un canal. Elle consiste à utiliser les noms des processus comme source et destination. Le format des primitives devient donc :

- `send` *msg* `to` *pcs*1

  Cette primitive envoie un message «msg» qui ne peut être reçu que par le processus possédant l'identificateur «pcs1».

- `receive` *message* `from` *pcs*2

  Cette primitive permet de recevoir un message dans «msg» qui provient du processus détenant l'identificateur «pcs2».

Ce type de désignation possède les propriétés suivantes :

- un canal est établi entre chacun des processus qui désire communiquer. La seule condition préalable est de connaître l'identificateur du processus ;
- un canal est associé à seulement deux processus ;
- il n'existe qu'un seul canal entre chaque processus ;
- le canal est bidirectionnel, i.e. les deux processus sont en mesure d'effectuer des «`send`» et des «`receive`» sur ce canal [1].

Ce mode de désignation est facile à implanter et à utiliser. Il permet à un processus de contrôler le moment exact auquel il reçoit un message de chacun des autres processus. C'est donc une **réception sélective**.

Le programme 5.1 présente notre petit système à traitement par lots, ré-implanté cette fois grâce à la communication par passage de messages avec la désignation directe. Cet exemple illustre bien le concept de pipeline. Un pipeline, tel qu'illustré à la figure 5.1 est une collection de processus concurrents dans laquelle la sortie de chacun des processus devient l'entrée d'un autre. L'information circule donc de façon similaire à celle d'un liquide dans un pipeline. Dans notre système, l'information passe du lecteur vers l'exécuteur puis vers l'imprimante. En fait ce mode de désignation est parfaitement adapté à l'implantation de pipelines.

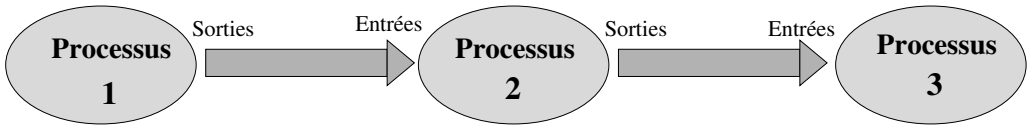

Figure 5.1 – Exemple de pipeline

Un autre interaction très importante que l'on retrouve fréquemment entre les processus est illustré à la figure 5.2. Elle est identifiée par l'expression «**clients/serveur**». Dans ce concept, un processus serveur rend des services à des processus clients. Ces derniers font des demandes de service en envoyant des message au serveur. Le serveur reçoit de façon répétitive les demandes, les exécute et retourne une réponse sous forme de messages aux clients.

La relation clients/serveurs est omniprésente dans tous les systèmes informatiques. Un processus qui contrôle une unité d'entrées/sorties est un serveur. L'unité en question peut être une imprimante par exemple. Tous les processus désirant accéder à cette unité (imprimer des données, résultats ou autres) lui envoient des commandes d'impression sous forme de messages. Le serveur reçoit la demande, l'exécute (imprime), retourne un message de fin et enfin traite la prochaine demande.

Les unités en charge de ces services sont des entités physiques ou virtuelles. En informatique, des serveurs sont déployés pour assurer de multiples fonctionnalités dont, en particulier, pour le

1. Un canal unidirectionnel signifie qu'un seul des deux processus peut faire des «`send`» et que l'autre ne peut faire que des «`receive`».

```
1  Program OPSYS;
2  process lecteur;
3      var ligne : entree;
4      loop
5         lecture ligne;
6         send ligne  to traitement;
7      end};
8  end process;
9  process traitement;
10     var ligne : entree;
11         resultat : sortie;
12     loop
13        receive ligne from lecteur;
14        traitement de ligne et génération de resultat;
15        send resultat to imprimante;
16     end;
17 end process;
18 process imprimante;
19     var resultat : sortie;
20     loop
21        receive resultat  from traitement ;
22        impression de} resultat;
23     end;
24 end process;
```

Programme 5.1 – Système à traitement par lots avec communication par messages

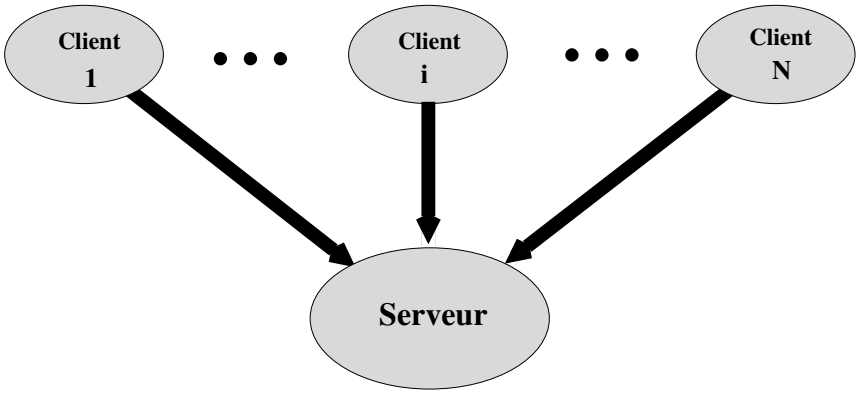

Figure 5.2 – Relation clients/serveur

Web, pour les transferts de fichiers (FTP, SFTP, FTPS), pour la gestion des adresses (DNS), pour la gestion des impressions, etc.

Ce qui est important dans cette relation, c'est que le serveur traite les messages dans l'ordre d'arrivée sans se préoccuper de sa provenance. En effet, les clients sont souvent inconnus du serveur et ce dernier doit faire une réception non sélective. Malheureusement, la désignation directe ne permet pas d'implanter facilement les interactions de type clients/serveur. En effet, la réception de message telle qu'elle a été spécifiée, force une réception sélective, i.e. dont la source est un processus précis. Cela fonctionne dans le cas d'un client unique. Mais dans ce cas, avouons que le serveur ne sert pas à grand chose ! ! Cette forme d'interaction n'est donc pas exploitable sous sa forme actuelle.

La solution proposée pour régler ce problème est de fournir une primitive de réception non sélective :

`receive` *message*

Ainsi, la désignation directe demeure mais on y ajoute une primitive très simple à implanter, qui répond au besoin de la relation clients/serveurs. Dans la plupart des environnements supportant la désignation directe, les deux formes de réception, sélective ou non sélective (parfois appelée symétrique ou asymétrique), sont fournies.

La facilité à implanter la désignation directe avec réception sélective ou non constitue un avantage certain. Elle possède toutefois quelques inconvénients :

- changement de processus ;

  La désignation directe ne supporte pas les changements de processus alors que de tels changements sont parfois requis du côté du serveur, par exemple, lors d'une mise à jour. L'usage de la désignation directe impliquerait de devoir modifier tous les clients qui communiquent avec ce dernier afin qu'ils utilisent le nouveau nom du serveur. Ce fait n'est ni commode, ni souhaitable.

- serveurs multiples ;

  Cette technique ne convient plus lorsque l'on veut implanter de multiples serveurs (voir la figure 5.3) pour un même service (comme le font Google, Facebook, Microsoft, etc) ou pour gérer des ressources identiques (comme de multiples imprimantes).

5.1.2 Boîte aux lettres (port global)

Pour contourner les deux inconvénients rencontrés avec la désignation directe, on opte pour un mécanisme plus général. Celui-ci est basé sur la notion de «porte globale», aussi appelée «boîte aux lettre» ou «désignation indirecte».

La boîte aux lettres apparaît dans l'énoncé `send` d'un processus, pour désigner la destination d'un message, et dans l'énoncé `receive` d'un autre processus, pour désigner la source du message. Les primitives adoptent donc le format suivant :

- `send` *message* `to` *boîte*

- `receive` *message* `from` *boîte*

Ainsi un message envoyé à une boîte aux lettres particulière peut être reçu par n'importe lequel des processus ayant accès à la boîte et ayant exécuté un énoncé `receive` sur cette même boîte.

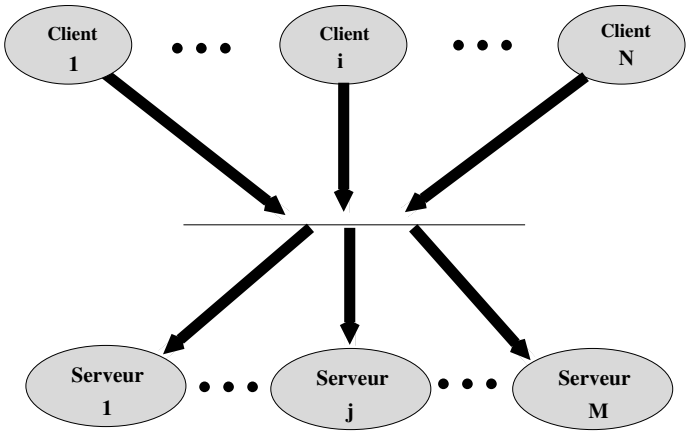

Figure 5.3 – Relation clients/serveur avec multiples serveurs

Ce mode de désignation convient parfaitement à une relation clients/serveurs. Tous les clients envoient leurs demandes à une boîte aux lettres particulière et tous les processus serveurs reçoivent leurs demandes de cette même boîte. La figure 5.4 décrit ce mode de fonctionnement. Chaque serveur ou type de serveurs possède sa propre boîte aux lettres qui identifie de façon unique les services qu'ils rendent. Les clients communiquent avec les serveurs de leur choix grâce à ces différentes boîtes aux lettres.

Selon le concept de portes globales, un canal de communication possède les propriétés suivantes :

- La communication implique l'existence d'une boîte aux lettres ;
  En effet, pour que la communication soit possible entre deux processus, une boîte aux lettres commune est requise. L'un des processus doit créer une boîte aux lettres pour que la communication s'établisse.
- Un canal peut s'associer à plus de deux processus ;
  Comme l'illustre la relation clients/serveurs présentée à la figure 5.4, plus de deux processus partagent le même canal.
- Plusieurs canaux peuvent exister entre une même paire de processus ;
  Il est possible d'établir plusieurs canaux de communication (boîtes aux lettres) entre deux processus. C'est utile, par exemple, quand un serveur souhaite envoyer des réponses à différents clients (voir la figure 5.5).
- Le canal est unidirectionnel ou bidirectionnel.

Un système implantant les boîtes aux lettres (ou portes globales) doit fournir un moyen de les identifier, de les créer et de les détruire.

Les portes globales sont très flexibles mais elles amènent certains défis lors de l'implantation. Il est nécessaire :

- d'implanter les opérations pour les manipuler telles que la création et la destruction.
- de déterminer à qui appartient la boîte aux lettres ;
  Elle appartient soit à un processus, soit au système sous-jacent. Dans le premier cas, un support du langage de programmation est requis et il est alors possible de distinguer le propriétaire des utilisateurs. Le propriétaire est le processus qui reçoit des messages de la porte

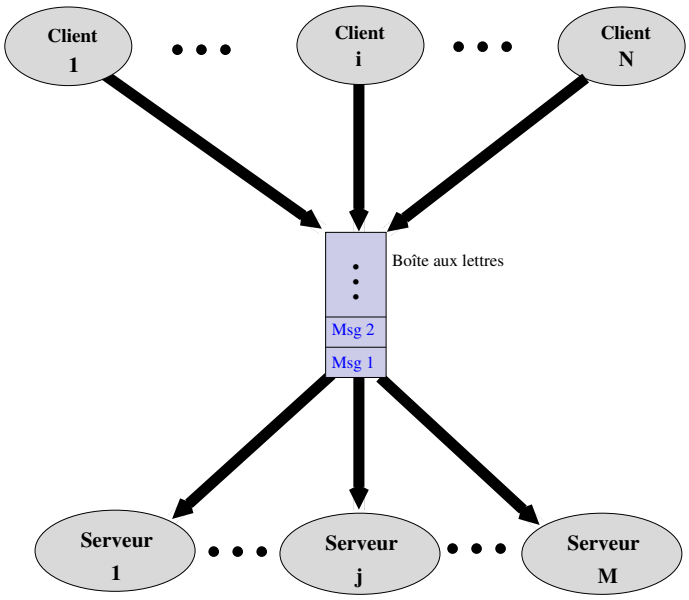

Figure 5.4 – Relation clients/serveurs avec multiples serveurs utilisant une boîte aux lettres

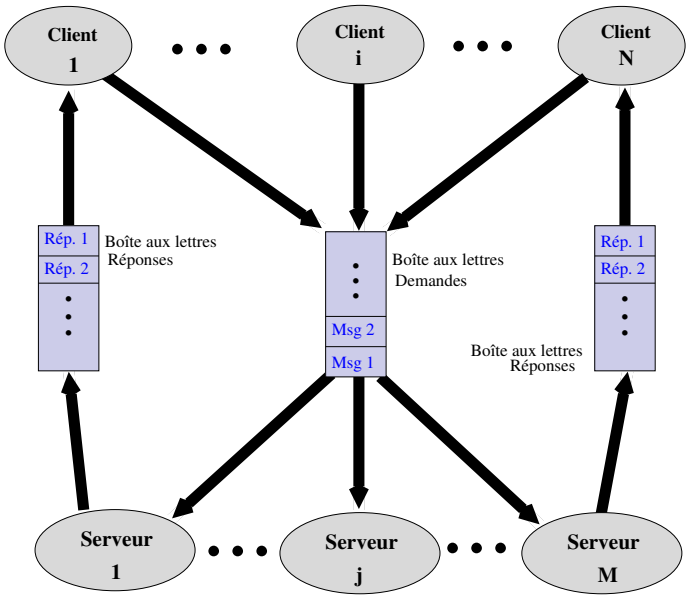

Figure 5.5 – Relation clients/serveur avec multiples boîtes aux lettres

globale et les utilisateurs sont ceux qui envoient des messages vers cette même porte. Comme chaque boîte ne possède qu'un seul propriétaire, il n'y a aucune confusion sur qui reçoit des messages. De plus, lorsque le propriétaire disparaît, la porte globale disparaît aussi. Lors d'un envoi subséquent à cette disparition, l'opération générera une erreur.
Il y a plusieurs façons de désigner le propriétaire et les utilisateurs d'une boîte aux lettres :

- Déclaration de variables ;
  Il est possible de permettre aux processus de déclarer une variable de type boîte aux lettres. Le propriétaire est le processus qui la déclare et les utilisateurs, eux, sont ceux qui connaissent l'identificateur de la variable.
- Déclaration externe ;
  Il est aussi possible que les boîtes aux lettres soient déclarées à l'externe des processus et partagées avec ceux-ci. Le rôle de chaque processus est alors spécifié séparément.

Le programme 5.2 présente la manière dont les variables de type boîte aux lettres sont déclarées dans le langage SR. Dans ce cas, le processus parent déclare la variable et tous les enfants l'utilisent sans restriction.

```
1  resource cs2()
2    # Déclaration de deux boites aux lettres : additionne et resultats
3    op additionne(id:int;val1:int;val2:int), resultats[3](rep:int)
4
5    process client(i := 1 to 3)
6      var reponse : int
7
8      write("Client ", i, "envoie sa demande")
9      send additionne(i, i+2, i+3)
10     receive resultats[i](reponse)
11     write("Réponse client ", i, " = >", reponse)
12   end
13   process server
14     var id : int, val1 : int, val2:int, rep : int
15     do true ->
16       receive additionne(id, val1, val2)
17       rep := val1 + val2
18       send resultats[id](rep)
19     od
20   end
21 end
```

Programme 5.2 – Boîte aux lettres en SR

Les boîtes aux lettres qui appartiennent au système d'exploitation sont gérés par ce dernier et ne sont pas attachées à un processus. Le systèmes d'exploitation fournit des primitives (appels système) pour :

- créer une nouvelle boîte aux lettres ;
- envoyer et recevoir des messages à une boîte aux lettres ;
- détruire une boîte aux lettres.

Le processus qui crée une boîte aux lettres est son propriétaire par défaut (si ce concept existe). Initialement le propriétaire est le seul processus autorisé à recevoir des messages de cette boîte. Toutefois le propriétaire a la possibilité de manipuler et de transférer les droits

sur la boîte aux lettres. La notion de propriété est donc transférable. De même, les droits d'envoi/réception sur la boîte aux lettres sont aussi modifiables (comme le sont les droits d'accès aux fichiers). Ainsi, le processus qui a créé la boîte a donc le loisir d'autoriser d'autres processus à accéder à celle-ci.
Parfois le système se charge lui-même de gérer les autorisations. Concernant le système Linux par exemple, les droits sur les boîtes aux lettres y sont gérés de la même façon que ceux sur les fichiers. Ainsi lorsqu'une boîte est créée, tous les processus de l'utilisateur possèdent les mêmes droits sur la boîte (read/write/execute). Les processus ont toutefois la possibilité d'accorder plus de droits aux membres du groupe ou aux autres processus (de la même manière que pour comme pour les fichiers).
Quand la boîte appartient au système, sa destruction doit être explicite. Si les processus oublient de la détruire, alors elle restera active dans le système. Aussi un ramasse-miette devient nécessaire.

Les permissions sous Linux

La protection de base sur les fichiers de Linux (et Unix en général) est assurée en divisant les utilisateurs en trois groupes (propriétaire, groupe de propriétaire et les autres) et les droits d'accès en trois types (lecture, écriture et exécution).
Ainsi, cette protection s'exprime avec 9 bits. Par exemple :

```
rwxr-xr-- .. zeus olympe ... mont
```

indique que le fichier `mont` appartenant à l'utilisateur `zeus` du groupe `olympe` est accessible en lecture/écriture/exécution (rwx) par `zeus`, en lecture et exécution (r-x) par les membres de son groupe et en lecture seulement (r--) par les autres utilisateurs.
Ces protections s'expriment aussi avec des valeurs octales. Ainsi la protection du fichier `mont` se représente aussi comme 754 (en octal). Dans la plupart des cas, les droits d'accès sur une boîte aux lettre sont 600 (en octal). La commande ipcs permet de voir les boîtes aux lettres disponibles ainsi que les droits d'accès qui leurs sont associées.

- Il faut gérer les accès simultanés ;
  Comme plusieurs processus sont en mesure d'accéder simultanément à la même boîte aux lettres et que ces processus sont potentiellement distribués sur plusieurs sites d'un réseau, l'accès à la boîte aux lettres exige une synchronisation complexe, fort coûteuse à implanter. C'est particulièrement le cas lorsque la boîte aux lettres est dupliquée sur plusieurs sites (multiples serveurs sur plusieurs ordinateurs différents) comme le met en évidence la figure 5.6.
  Les questions suivantes sont alors soulevées :
  - Que faire lors d'un envoi sur une boîte distribuée ?
    Dans cette situation, les messages envoyés à la boîte se doivent d'être visibles sur tous les sites visés. Un serveur centralisé est envisageable bien qu'une source potentielle de congestion. Il est aussi possible de créer une copie de la boîte aux lettres sur chacun des sites et d'envoyer à chacune une copie des messages. La gestion de toutes ces copies devient alors complexe.
  - Que faire lors de la réception sur une boîte distribuée ?
    Lors de la réception, il faut s'assurer que le message est reçu par un seul processus.

L'implantation est plus simple si on a recourt à un serveur central. Toutefois, s'il y a plusieurs copies de la boîte aux lettres, une synchronisation est requise. Ainsi, après la réception, il faut détruire toutes les copies sur tous les sites pour éviter qu'un message particulier ne soit reçu plus d'une fois.

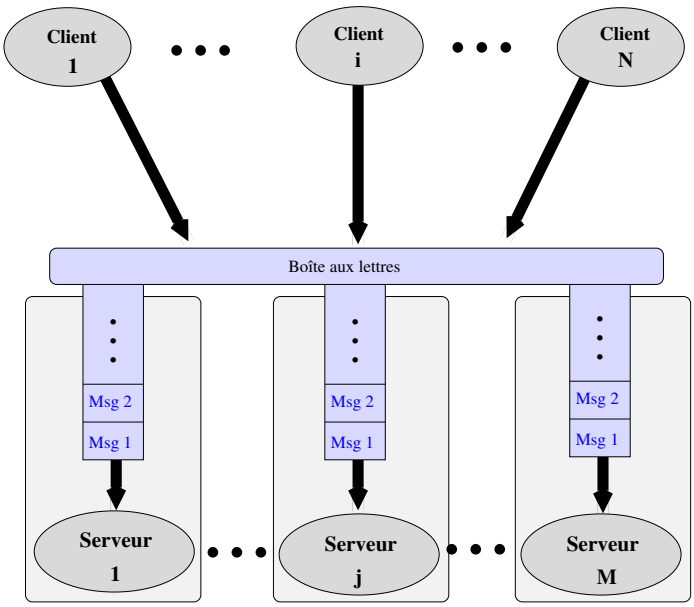

Figure 5.6 – Relation clients/serveurs avec multiples sites

Boîte aux lettres : manipulation

Exemple de code à venir.

Boîte aux lettres du système

Exemple de code à venir.

5.1.3 Port

Dans certaines circonstances, comme déjà brièvement mentionné, on utilise un cas particulier des boîtes aux lettres, appelé «port», où un seul processus est autorisé à recevoir des messages d'un port.

Due à cette restriction, les ports sont plus simples à implanter que les boîtes aux lettres. En effet, comme toutes les opérations de réception qui désignent un port n'apparaissent que dans un unique processus (donc sur le même site), la gestion des accès au port est facilitée (aucune coordination nécessaire lors de l'accès aux multiples copies sur différents sites).

Le concept de port est une solution simple et efficace pour la mise en œuvre de la relation clients/serveur (avec un serveur unique).

Ce type d'adressage ressemble à la désignation directe avec réception non sélective mais il est plus souple. Il permet en effet de changer le nom du processus serveur sans que cela n'affecte les clients. Cela offre aussi la possibilité à des processus d'accéder à plusieurs ports.

La notion de port est donc un cas particulier de la boîte aux lettres et la désignation directe est un cas particulier du port.

5.1.4 Utilisation des ports

Deux usages distincts sont associés aux portes :

- orientation vers les messages ;
  Dans ce cas plusieurs processus sont autorisés à envoyer des messages vers un même port. C'est ce mode qui est choisi par défaut dans tous nos exemples.
- orientation vers les liens ;
  Dans ce cas, un seul processus est autorisé à envoyer des messages à un port. Un canal unique privé est alors créé entre deux processus.

5.1.5 Désignation des ports (statique ou dynamique)

Une dernière difficulté à résoudre lorsqu'on recourt aux ports ou aux boîtes aux lettres : celle reliée au «moment» où les processus apprennent les noms des ports avec lesquels ils désirent communiquer. Les deux solutions sont :

- La désignation statique
  Dans ce cas, le code du programme contient le nom du port au moment de sa compilation. Si le nom du port change, alors il faudra modifier le code et recompiler le programme.
- La désignation dynamique ;
  Avec ce mode, les processus ignorent le nom du port au début de leur exécution. Pendant leur exécution, un protocole spécial leur permet d'apprendre le nom du port recherché. De nombreux modes de fonctionnement ont cours à ce niveau. Par exemple, la diffusion d'un message pour qu'un autre processus sur le réseau «écoute» et fournisse l'information. Ou encore, faire appel à un «*serveur de nom*» dont le rôle est de distribuer les noms des ports associés à des services.
  Les serveurs DNS sont un exemple de «serveurs de nom». En Java, le service `Jini` fournit un serveur de nom.

5.2 Synchronisation dans les messages

La transmission du message assure automatiquement la synchronisation. Il existe toutefois différents degrés de synchronisation, souvent reliés à la capacité du système d'emmagasiner ou non les messages

5.2.1 Synchronisation (Envoi)

Les différents degrés de synchronisation lors d'un envoi sont :

- le processus expéditeur bloque jusqu'à la réception du message (aucun emmagasinage, «rendez-vous») ;

Cette situation se produit lorsque l'environnement n'emmagasine aucun message. Le processus expéditeur se bloque jusqu'à ce que le destinataire reçoive les messages ou bien se voit retourner un code d'erreur si le destinataire n'est pas prêt.
Ce mécanisme s'appelle «**passage de messages synchrones**» ou «**rendez-vous**». Dans ce mode de communication, le système fournit un canal de communication sans aucune capacité d'emmagasinage.

- Le processus expéditeur ne bloque jamais (espace infini) ;
  Cette situation se produit lorsque le système emmagasine les messages dans un espace supposé infini. L'expéditeur ne se bloquera donc jamais.
  Certains systèmes fournissent des commandes spéciales pour connaître l'état d'un message, à savoir s'il est reçu ou non. Cela s'avère parfois nécessaire car contrairement au mécanisme de rendez-vous, la poursuite de l'exécution de l'émetteur ne signifie pas que le message a été reçu.
  Ce type de synchronisation est appelé **passage de messages asynchrones**.
- Le processus expéditeur bloque selon la régulation du flux des messages ;
  En pratique, il n'existe aucun espace infini pour emmagasiner les messages. Il faut donc traiter le cas où tout l'espace alloué est occupé. Il est alors question de la **régulation du flux des messages**.
  Cette régulation se base sur un **nombre maximum de messages** :
  - en attente par **processus** ;
  - en attente par **portes** ;
  - en circulation dans tout le **système**.

  Dans tous les cas, lorsque le maximum est atteint le système bloque chacun des processus qui tente un envoi ou leur retourne un code d'erreur.
- Le processus expéditeur se bloque jusqu'à la réception d'une réponse ;
  Il arrive qu'un processus qui envoie un message soit bloqué jusqu'à l'arrivée d'une réponse. Cette sémantique est celle des RPC (*Remote Procedure Call*) que l'on retrouve dans plusieurs environnements modernes (`Java/RMI, Python/Pyro`).

5.2.2 Synchronisation (Réception)

Pour le récepteur d'un message, le blocage ne dépend pas de la capacité du système à emmagasiner les messages.

Les différents degrés de synchronisation lors de la réception d'un message sont les suivants :

- Le récepteur se bloque jusqu'à l'arrivée du message ;

  C'est le mode de blocage normal et par défaut dans la plupart des environnements. Lors d'une réception, l'émetteur se bloque jusqu'à ce que le message attendu se présente.

  Certains environnements fournissent une primitive qui retourne l'état de la file pour éviter un blocage.

- Le récepteur ne se bloque pas ;

  Lors d'une réception, aucun blocage ne se produit. L'opération retourne soit avec le message, soit avec un message d'erreur indiquant qu'il n'y a aucun message en attente de réception. Dans ce cas, il faut prévoir une primitive différente pour attendre l'arrivée d'un message. MPI offre ce type de primitive.

5.3 Nature des messages

Quelques autres caractéristiques associées aux messages sont aussi susceptibles d'influencer la complexité du système de communication.

5.3.1 Les messages typés

Il est fréquent que les messages soient typés, par exemple, ceux de type demande ou de type réponse.

L'assignation d'un type à un message se fait :

- soit avec un champs spécial contenu dans le message ;
- soit avec une commande spéciale (du genre sendReply) ;
  Le système RC4000 conçu par Brinch Hansen[18, 19, 41] offrait des primitives différentes pour typer les messages.
- soit avec des ports particuliers.
  Il est effectivement courant de faire appel à des ports pour les demandes et à d'autres ports pour les réponses.

5.3.2 La taille fixe ou variable des messages

La taille, fixe ou variable, des messages influence directement la complexité du système.

Ainsi, si la taille des messages est fixe, l'implantation du système est simple. Ce choix facilite aussi son utilisation car il n'est pas nécessaire de s'enquérir de la longueur du message avant de lui réserver l'espace utile à son emmagasinement.

Autrement, si la taille des messages est variable mais contrainte à une certaine longueur maximale, l'implantation et l'utilisation du système deviennent plus complexes. Par contre, ce choix a l'avantage de préserver une grande quantité de bande passante, puisqu'on évite ainsi de transmettre des champs quasi-vides ou entièrement vides lors de certaines communications. En effet, si la taille des messages est fixe, (supposons 4k) alors chaque message consommera 4k de la bande passante même si le contenu réel n'est que d'un seul caractère (genre ACK).

5.3.3 Les différents formats des messages

Tous les messages dans un système de communications par messages ont un format précis. Certains systèmes définissent un format unique et d'autres plusieurs formats distincts.

Généralement, plus le système supporte de types de messages, plus il doit supporter de formats distincts. Si ce n'est pas le cas, le format unique devra contenir tous les champs possibles et ceux-ci seront transmis avec chaque message même si cela s'avère inutile.

En fait, ces trois domaines (type, taille et format) sont interreliés. Ainsi des messages de longueurs variables influencent les formats qui eux, sont influencés par les types. Plus on supporte de types, plus le nombre de formats est élevé. Plus les formats sont nombreux, plus les variations de tailles pour les messages sont importantes.

5.4 Protection

Un système orienté vers le passage de messages doit se protéger des influences indirectes telles que :

- des messages erronés ;

  Une technique pour éviter les messages erronées consiste à les filtrer. Pour y parvenir, on se doit de définir une politique d'autorisations qui gère les permissions d'envoi de messages d'un processus vers un autre processus. Certaines politiques se basent sur l'arbre de création de processus. Par exemple :

  – Un processus reçoit exclusivement des messages de ses enfants ;
  – Un processus reçoit exclusivement des messages de ses descendants ou des descendants de son parent.

  Il est aussi possible de se baser sur une matrice d'accès pour filtrer les messages (liste d'accès ou pouvoirs).

- la fin de processus ;

  Si l'un des processus interlocuteurs termine de façon impromptue, les situations suivantes sont susceptibles de se produire :

  – un processus attend un message d'un autre processus qui n'existe plus.
    Le système doit se protéger contre ce genre d'événement et éviter que l'expéditeur attende indéfiniment. La solution standard consiste à associer une horloge de garde (*timeout*) à chaque opération de communication. Lorsque le temps maximal d'attente est écoulé, un message d'erreur est retourné au processus en attente et ce dernier poursuivra son exécution.
  – Un processus envoie un message à un processus qui n'existe plus ;
    Si le système emmagasine les messages, il n'y a pas de problème pour l'expéditeur (sauf s'il attend une réponse). Le système doit toutefois éliminer les messages sans destinataire. S'il détecte que ces derniers n'existent plus au moment de l'envoi, il retourne un code d'erreur et n'emmagasine pas le message. Si c'est impossible, le système devra de temps à autre exécuter un «ramasse-miettes» pour détruire les messages orphelins.
    Dans le cas contraire où il n'y a aucun emmagasinage (rendez-vous), le système devra mettre en place une horloge de garde pour éviter une attente infinie. Il est aussi possible que, dans le cadre du «ménage» fait lors de la mort du processus, l'on puisse détecter et débloquer les processus en attente.

- les erreurs de communication ;

  Les systèmes orientés messages sont particulièrement populaires sur les réseaux d'ordinateurs où les messages circulent sur une voie de communication non fiable. Parfois, la tentative de communication échoue. Les différentes causes de ces échecs sont :

  – la destruction du correspondant (cas déjà abordé) ;
  – la perte de messages ;
    Pour détecter les pertes de messages, tout comme celle d'un correspondant, les horloges de garde sont fréquemment privilégiées. Lorsqu'un message est considéré perdu, il est généralement retransmis.

  - la violation du protocole de communication ;
  - la duplication de messages ;
    La retransmission d'un message supposé perdu, peut engendrer de multiples copies du même message. On se doit alors de détecter ces cas et d'éliminer les copies redondantes.
  - la corruption du contenu d'un message ;
    Le contenu d'un message est susceptible d'être altéré par erreur (bruit sur la ligne ou autres). Encore là, la détection de cette corruption et la mise en œuvre d'actions conséquentes sont primordiales. Il est possible de recourir à des codes détecteurs d'erreur (bit de parité par exemple) et de retransmettre le message lorsqu'une corruption est détectée. Dans certaines situations, il est carrément impossible de retransmettre (messages qui proviennent de Mars par exemple). Dans ces cas, des codes correcteurs d'erreurs sont employés (tel Hamming et autres).

Pour détecter toutes les erreurs possibles, il est important de répondre à deux questions :

- Qui est en charge de détecter l'erreur ?
  Il y a trois situations possibles :
  - le système sous-jacent détecte l'erreur et effectue la reprise.
    Par exemple, si une perte de message est détectée, le système se charge de le retransmettre sans que l'application en soit informée.
  - le processus détecte l'erreur et effectue la reprise ;
    Dans ce cas, l'application elle-même a la charge de détecter les erreurs et d'effectuer elle-même la reprise. Ici, le système sous-jacent offre un service de base sans aucune garantie.
  - le système sous-jacent détecte l'erreur et le processus effectue la reprise ;
    Dans ce dernier cas, le système met en place les mécanismes nécessaires à la détection des erreurs et lorsque cela se produit, il informe l'application. Celle-ci doit elle-même effectuer la reprise.

  Il est possible que ces trois méthodes de fonctionnement pour la détection et la reprise soient utilisées conjointement mais c'est alors pour différents types d'erreurs.
- Comment détecter l'erreur ?
  - horloge de garde ;
    L'horloge de garde s'avère être l'outil le plus couramment employé pour détecter notamment certains types d'erreurs impliquant une attente trop longue (perte de messages, perte de correspondants, ...).
  - ramasse-miettes ;
    Les ramasse-miettes servent à libérer l'espace occupé par des messages orphelins.
  - numéros de version
    Les numéros de version sont déployés pour détecter des copies redondantes d'un message.
  - code détecteur et correcteur ;
    Les bits de parité, code de Hamming et autres techniques sont utilisés pour identifier la corruption dans les messages.
  - Autres...
    D'autres techniques existent mais sont plutôt reliées au domaine de la tolérance aux fautes. Nous aborderons ce sujet ultérieurement.

5.5 Implantation

Comment un système de communication par messages s'implante-t-il ?

Un système de communication par messages n'est qu'une autre appellation du problème des producteurs/consommateurs. Il est donc généralement implanté par un ensemble de tampons contenant les messages produits (`send`) et consommés (`receive`). L'emploi d'une liste chaînée de type FIFO est typique pour la mise en place de ces tampons.

Il est d'usage courant d'intégrer les systèmes de communication par messages au niveau du noyau du système d'exploitation. À ce niveau, l'implantation dépend de la méthode de désignation choisie. Toutefois, l'information est fortement reliée aux descripteurs des processus (PCB) et à la capacité du système à emmagasiner des messages.

5.5.1 Désignation directe et emmagasinage de messages

Dans le cas où les systèmes emmagasinent les messages et que la désignation directe est adoptée, on associe tout simplement à chaque descripteur de processus une liste chaînée de messages en attente de réception comme l'illustre la figure 5.7.

Une liste de messages emmagasinés constitue une ressource partagée. Puisque les expéditeurs et le récepteur ont accès à cette liste, générant ainsi un éventuelle «**condition de course**», l'accès est alors contrôlé par un sémaphore d'exclusion mutuelle.

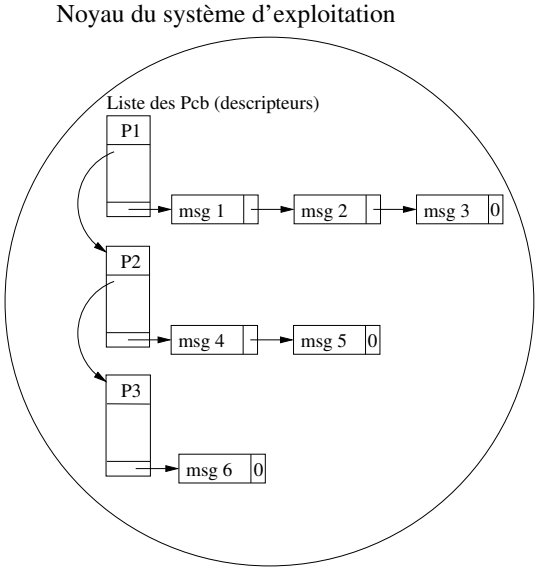

Figure 5.7 – Liste de messages avec désignation directe et emmagasinage.

5.5.2 Désignation indirecte et emmagasinage de messages

Si le système intègre le concept de ports ou de boîtes aux lettres, il doit fournir une structure pour supporter ces concepts. Une table sert typiquement à implanter les ports. Chaque entrée de

la table représente un port. Chacune d'elles contient l'information nécessaire pour sa gestion ainsi qu'une liste des messages en attente de réception. L'indice dans la table représente le numéro du port utilisé dans les opérations de communication. La figure 5.8 décrit une implantation possible de cette table. La commande pour recevoir un message (`msg4`) se rédige comme suit : `receive` *`mon_message`* `from 2`.

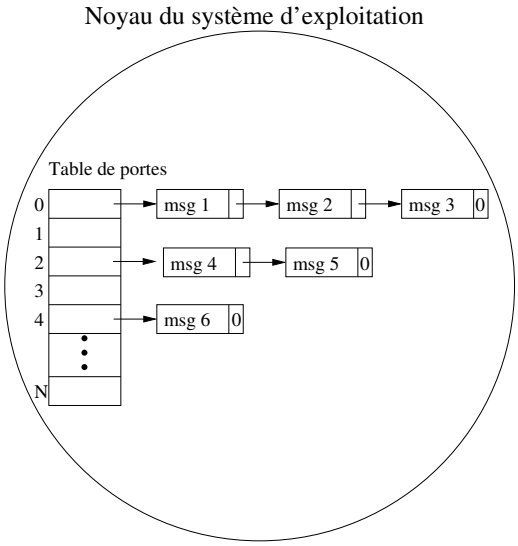

Figure 5.8 – Liste de messages avec désignation indirecte et emmagasinage.

5.5.3 Aucun emmagasinage de message (rendez-vous)

Si le système n'emmagasine aucun message, il doit maintenir une table de rendez-vous. Cette table contient les noms des processus en attente pour la réception ou l'envoi d'un message. Lorsque la source et la destination d'un envoi de message correspond à la cible et à la source d'une commande de réception respectivement, alors le système effectue l'échange.

La figure 5.9 présente un exemple de table. Ainsi, lorsque la commande suivante est émise par le processus `P2`,

`P2 :: receive` *`mon_message`* `from P1`,

l'échange s'effectue car la table contient un envoi de message de `P1` vers `P2`.

5.5.4 Transmission des messages

Un dernier élément à considérer lors de l'implantation d'un système de communication par messages est celui du mode de transmission du message. Voici les deux façons de transmettre les messages :

- par contenu

Émetteur la commande	Cible de la commande	Commande
P1	P2	`send`
P3	P4	`receive`
P5	P6	`send`
P7	P8	`send`
P9	P10	`receive`

Figure 5.9 – Table de rendez-vous.

Selon ce mode de transmission, on recopie tout le message à chacune des transmissions. Cela entraîne certes une surcharge importante sur un système centralisé mais, il est dès plus efficace sur un système réparti (réseau) où il n'y a aucun partage de mémoire.

- par adresse
  En procédant avec une transmission par adresse, on ne recopie pas le message au moment de la transmission. On envoie seulement une référence vers lui. Cette approche fonctionne dans un environnement centralisé dans lequel les processus partagent de la mémoire. On évite ainsi de recopier des messages entiers. Elle s'avère toutefois difficile à réaliser sur un système distribué où il n'y a aucun partage de mémoire.
  Un risque associé à cette approche est celui d'écraser un message par l'envoi d'un autre avant la réception du premier qui sera alors perdu. Pour éviter cette situation, soit qu'un processus s'assure de la réception du message avant l'envoi d'un autre, soit qu'il réserve un nouvel espace mémoire spécifique pour emmagasiner le second message.
  MPI fournit ce type de transmission de messages.

5.6 Les variables de type *futures/promesses*

Les variables de type `futures` et `promesses` sont des concepts servant à la synchronisation et à la communication dans certains langages récents. Une variable de type `future` est définie comme une variable dont le contenu sera éventuellement disponible. Elles permettent d'écrire des fonctions asynchrones.

En fait, les variables de types `future` et `promesse` sont des concepts très similaires aux ports de communication (elles sont en fait des ports de communication soumis à certaines restrictions). Toutefois ce type de variable est à «assignation unique» (ou à usage unique) et possède au moins deux états : «complété» ou «indéterminé». Une lecture d'une variable de type `future` à l'état «indéterminé» est bloquante (tout comme pour la réception d'un message).

Le mode de fonctionnement des variables de type future est le suivant :

- La déclaration de variable «`future X`» retourne immédiatement une variable de type «`future`» pour éventuellement contenir la valeur de l'expression `X`;

- L'évaluation de `X` est lancée en parallèle (fil ou événement) ;
- Lorsque l'évaluation de `X` termine, sa valeur est «emmagasinée» dans la variable de type `future` et est disponible pour une «lecture».

Les programmes 5.3, 5.4, 5.5 et 5.6 sont des exemples d'utilisation de variables de type `future` et `promesse` en Scala et en C++. Pour plus d'informations, se référer à l'annexe A.

```
1 val f = Future { Http("http://.....").assString  }
2
3 f onComplete {
4     case Succes(response) => println(response.body)
5     case Failure(t) => println(t)
6 }
```

Programme 5.3 – Exemple d'utilisation de variables de type future en Scala

```
1 #include <thread>
2 #include <iostream>
3 #include <future>
4
5 void fonction1()
6 {
7    std::cout << "Fil d'execution..." << std::endl;
8 }
9 int main()
10 {
11     auto future1 = async(launch::async, fonction1);
12     future1.get();
13     cout << "Le futur est arrive ..." << endl;
14     return 0;
15 }
```

Programme 5.4 – Exemple d'utilisation de variables de type future en C++

```
1 #include <thread>
2 #include <iostream>
3 #include <future>
4 using namespace std;
5
6 int main()
7 {
8    auto future1 = async(launch::async, [](){cout << "Fil 1" << endl;});
9    future1.get();
10    cout << "Le futur est arrive ..." << endl;
11    return 0;
12 }
```

Programme 5.5 – Exemple d'utilisation de variables de type future en C++

```
\begin{lstlisting}
#include ...
using namespace std;

int main(){
    auto promesse = promise<std::string>();
    auto producteur = thread([&]
                        {
                              promesse.set_value("Bonjour.....\n ");
                        });

    auto futur1 = promesse.get_future();
    auto consommateur = thread([&]
                          {
                             std::cout << futur1.get();
                          });
    producteur.join();
    consommateur.join();
    return 0;
}
```

Programme 5.6 – Exemple d'utilisation de variables de type future et promise en C++

5.7 En pratique ...

5.7.1 Système d'exploitation

La plupart des systèmes d'exploitation offrent la possibilité de communiquer par messages, autant localement qu'à distance. Localement, la communication résulte de fonctionnalités propres au système. En Unix (Linux et autres), il est possible d'adopter les files de messages (message queue ou les messages aux normes Posix). [2]

En ce qui concerne la communication en réseau, tous les systèmes offrent une interface communément appelée «*socket*» pour permettre la communication inter-machines.

5.7.2 *Middleware*

Plusieurs «*middleware*» ont été développés avec pour objectif de fournir des environnements de communication par messages. Parmi eux, MPI (Message Passing Interface) fournit un système de communication inter-processus pour les environnements de grappes de calcul (calcul parallèle et scientifique).

Les MOM, eux, sont des environnements qui dispensent la communication par messages entre applications sur un réseau d'ordinateur. Plusieurs entreprises telles que Microsoft, IBM fournissent de tels environnements.

5.7.3 Langages

Certains langages implantent la communication par message, localement ou à distance, par l'intermédiaire de bibliothèques. Certains, comme SR et JR, l'intégraient dans le langage. Plus récemment, d'autres langages procurent cette facilité par l'intermédiaire des variables de type `future` et *promesse* intégrées, elles aussi dans le langage.

2. Ajouter les messages de Windows.

Annexe A

Futures et promesses

Ces concepts ne sont pas récents et sont issus de la programmation fonctionnelle ou autre paradigme similaire (logique).

Les variables de type *future* (*futur*) et *promesse* (*promise*)[1, 25, 32, 36, 40] consistent en des abstractions adoptées par plusieurs langages modernes comme outil de synchronisation et de communication. Les termes *future*, *promesse* ou autres (*delayed*, *deferred*, ...) sont généralement interchangeables, au sens qu'ils jouent exactement le même rôle (ce sont des synonymes, bien que certains langages fassent une distinction entre eux).

L'élément le plus important à retenir de ces notions est que ce sont des outils de communication et de synchronisation qui facilitent la programmation concurrente. Les avantages de ces constructions sont qu'elles permettent l'exécution d'opérations à la chaîne, On dit alors d'elles qu'elles sont composables. Il est donc envisageable d'attacher une fonction de rappel ou une continuation à la résolution d'une variable de type *future*.

A.1 `Future` et `promesse` comme des concepts interchangeables

Selon les auteurs et les langages, les définitions des types *future* et *promesse* varient. Lorsque les variables de type *future* et *promesse* sont tenues pour synonymes, c'est qu'elles sont considérées comme des conteneurs hébergeant éventuellement une valeur. Ainsi, la construction « *(future X)* » retourne immédiatement un conteneur pour la valeur de l'expression *X*. Dans certaines situations, elle amorce aussi concurremment l'évaluation de *X*. Quand l'évaluation de *X* terminera, la valeur sera placée dans le conteneur.

Une autre façon de visualiser ce concept consiste à lui associer une certaine notion de temps. Ainsi, on suppose que la valeur de l'objet diffère d'états selon le moment où elle est lue. La conceptualisation la plus simple est d'associer à l'objet deux états qui dépendent du temps :

1. l'état «complété»

   Le calcul est complété et la valeur est disponible ;
2. l'état «indéterminé»

   Le calcul n'est pas encore complété et la valeur n'est pas disponible.

Notons que d'autres états pourront s'ajouter pour la prise en main des erreurs, l'annulation ou autres événements.

Une variable de type *future* ou *promesse* est un objet à assignation unique susceptible d'être typé ou non .

Certaines implantations sont bloquantes (synchrones) d'autres non (asynchrones). Pour les cas asynchrones, deux approches existent pour démarrer l'exécution du calcul, soit explicitement ou implicitement.

Javascript [11, 13] supporte ce type de variable seulement sous le nom de `promise`. L'objet de type `promise` (pour *promesse*) est utilisé pour réaliser des traitements de façon asynchrone. Une variable de type *promesse* représente en une valeur disponible soit maintenant, soit dans le futur, voire jamais. La variable de type `promise` s'emploie souvent avec la déclaration de fonction «async» [9] qui définit une fonction dite «asynchrone», soit une fonction qui s'exécute de façon asynchrone et ce, grâce à la boucle d'évènements. À la fin de l'exécution de la fonction, le résultat (valeur de retour) est retourné par l'intermédiaire de la variable de type `promise`.

A.2 `Future` et `promesse` comme des concepts distincts

Lorsqu'il y a une distinction à faire entre ces deux termes, une variable de type *future* est alors toujours définie en tant que valeur éventuellement disponible. C'est donc une abstraction pour permettre la récupération d'un résultat «à venir», puisque celui-ci ne sera disponible ou évalué qu'ultérieurement. C'est une variable accessible uniquement en lecture au moment où l'information sera disponible. Quant à elle, la variable de type *promesse*, sera perçue comme une variable dans laquelle un résultat sera éventuellement écrit et rendu disponible dans la variable de type *future*. L'action permettant d'attribuer sa valeur à une variable de type *promesse* est généralement qualifiée par les termes résolution, fixation ou réalisation.

En fait, ces deux types de variables sont les deux extrémités d'un port de communication unidirectionnel. Ainsi la variable de type *future* représente l'extrémité du port de communication à laquelle la réception s'effectue et la variable de type *promesse* représente l'extrémité à laquelle l'envoi se fait.

Ces types de variables, tout comme les ports de communication, permettent aux développeurs de relier ensemble des fonctions asynchrones leur accordant ainsi le loisir de se synchroniser et de s'échanger de l'information.

De nombreux langages offrent deux structures distinctes aux variables de type *future* et *promesse*. Les langages tels Java, C#, C++ et Scala font partie de ce groupe. Dans ce cas, une variable de type *future* est une référence, en lecture seulement, à une valeur qui n'a pas encore été calculée et elle est associée à une variable de type *promesse*, qui en est une à assignation unique. En d'autres mots, une variable de type *future* est une fenêtre, en lecture seulement, sur une valeur écrite dans une variable de type «`promesse`». Aux sections qui suivent, nous présentons des exemples en C++ illustrant ce mode de fonctionnement.

En Scala, les variables de type *future* sont naturellement asynchrones. En java, pour les versions précédant la version 7, les variables de type *future* étaient synchrones. Puis, à partir de java 7, les variables de type *future* furent asynchrones.

A.3 Utilité

Les variables de type *future* et *promesse* sont devenus populaires avec l'arrivée en force des systèmes parallèles et distribués. Elles sont maintenant couramment utilisées et disponibles dans des langages tels JavaScript, Scala, Java, C++, ... Elles sont utiles lorsque des systèmes communiquent et qu'il y a un risque important de latence [1] lors des échanges d'information. De façon générale, on identifie les secteurs suivants comme des contextes favorables à ces types de construction :

1. Les modes requêtes/réponses (appel de service) ;
   Une variable de type *future* représente la réponse à la requête.
2. Une entrée/sortie ;
   Une variable de type *future* est fournie pour représenter l'appel à une opération d'entrée/sortie et son résultat.
3. Un long calcul ;
   Une variable de type *future* est l'outil parfait pour recevoir le résultat attendu et camoufler la latence d'un long calcul.
4. Les requêtes à une base de données (semblable aux requêtes/réponses) ;
5. Les RPC ;
   Tout comme pour un long calcul, les variables de type *future* servent aussi à implanter des RPCs asynchrones et à masquer la latence du réseau.
6. La lecture de données sur un port de communication ;
7. Le traitement des horloges de garde.

A.4 Implantation

Diverses implantations des variables de type *future* et *promesse* sont disponibles. L'objectif de chacune d'elles est d'exploiter au mieux les ressources. Les deux principales techniques pour ce faire sont : les fils d'exécution et les événements.

On qualifie l'usage de ces types de variables d'implicite ou d'explicite. À noter que ces termes n'ont toutefois pas le même sens selon les auteurs.

A.4.1 Les fils d'exécution vs la boucle événementielle

Les fils d'exécution

Pour effectuer les tâches qui produiront les résultats des variables de type *future*, certains environnements adoptent des fils d'exécution. Certains, tels `Scala` et `Java`, assignent un groupe de fils d'exécution (Thread Pools) à cette tâche. En `Scala`, ce groupe doit être passé en paramètre (explicite ou implicite) à la variable. Dans d'autres environnements, tel C++, un nouveau fil d'exécution est créé pour répondre à chaque variable de type *future* (création de fils ou `async`).

Les programmes suivants présentent des exemples d'applications des variables de type *future* et de *promesse* en `Scala` et en `C++`. Les programmes A.2 et A.3 sont identiques à un détail près

1. La latence est une mesure de délai. C'est la période de temps entre le moment où une demande est faite et celui ou la réponse arrive.

puisque dans le second exemple, on recourt à une fonction lambda.

```
val f = Future {
Http("http://api.fixer.io/latest?base=USD").assString
}

f onComplete {
    case Succes(response) => println(response.body)
    case Failure(t) => println(t)
}
```

Programme A.1 – Exemple d'utilisation d'une variable de type «`future`» en `Scala`

```
#include <thread>
#include <iostream>
#include <future>
#include <iostream>

using namespace std;
void fonction1()
{
    cout << "Je suis un fil d'execution......" << endl;
}
int main(){
    auto future1 = async(launch::async, fonction1);

    future1.get();
    cout << "Le futur est arrivé ..." << endl;
    return 0;
}
```

Programme A.2 – Exemple d'utilisation d'une variable de type «`future`» et de l'énoncé `async` en `C++`

Boucle événementielle (Event Loops)

`JavaScript` et `Python` (`asyncio`) choisissent plutôt la boucle événementielle pour implanter les variables de types *future* et *promesse*. La création d'une de ces variable n'implique donc pas la création d'un fil d'exécution, mais plutôt celle d'un événement qui sera éventuellement traité par la boucle événementielle. Dans ce cas précis, c'est l'utilisation d'entrées/sorties asynchrones qui augmente le parallélisme.

L'introduction au langage `Javascript` de Jake Archibald [2] fournit des exemples recourant aux variables de type *promesse*.

A.4.2 Exécution implicite ou explicite

Accès implicite ou explicite

Selon Wikipedia [40], l'utilisation des variables de type *future* est soit implicite ou soit explicite, faisant ainsi référence à la méthode utilisé pour l'obtention du résultat. Ainsi à chaque usage d'une

```cpp
#include <thread>
#include <iostream>
#include <future>
#include <iostream>

using namespace std;
int main(){
    auto future1 = async(launch::async,
                         [](){cout << "Je suis un fil d'execution" << endl;});

    future1.get();
    cout << "Le futur est arrivé ..." << endl;
    return 0;
}
```

Programme A.3 – Exemple d'utilisation d'une variable de type «`future`» et de l'énoncé `async` en C++ (fonction lambda)

```cpp
#include <thread>
#include <iostream>
#include <future>
#include <iostream>
#include <chrono>
#include <vector>

int main(){
    std::vector<std::future<size_t>> mes_futurs;

    for (size_t i = 0; i < 10; ++i) {
        mes_futurs.emplace_back(std::async(std::launch::async, [](size_t param){
            std::this_thread::sleep_for(std::chrono::seconds(param));
            return param;
        }, i));
    }
    std::cout << "Debut test de fin" << std::endl;
    for (auto &future : mes_futurs) {
      std::cout << future.get() << std::endl;
    }
    return 0;
}
```

Programme A.4 – Exemple d'utilisation d'une variable de type «`future`», de l'énoncé `async` et du type `vector` en `C++`

```cpp
#include <thread>
#include <iostream>
#include <future>
#include <iostream>
#include <chrono>
#include <vector>

int main(){
    auto promesse = std::promise<std::string>();
    auto producteur = std::thread([&]
    {
        promesse.set_value("Bonjour.....\n ");
    });
    auto futur1 = promesse.get_future();
    auto consommateur = std::thread([&]
    {
        std::cout << futur1.get();
    });
    producteur.join();
    consommateur.join();
    return 0;
}
```

Programme A.5 – Exemple d'utilisation de variables de types «`future`» et «`promesse`», ainsi que des fils d'exécution en `C++`

variable de type «*future*» implicite, on obtient automatiquement sa valeur de la même manière qu'avec une variable normale (une assignation). Dans la cas d'une référence à une variable de type *future* explicite, on doit explicitement appeler une fonction pour obtenir sa valeur, telle la méthode `get` de la bibliothèque `java.util.concurrent.Future` en `Java`. Les variables de type *future* explicites sont implémentés dans une bibliothèque, tandis que variables de type «*future*» implicites nécessitent un support direct du langage.

Démarrage implicite ou explicite

Selon l'approche implicite, il n'est pas nécessaire d'amorcer explicitement le calcul. La création de la variable de type *future* ou *promesse* démarre immédiatement le calcul.

En revanche selon l'approche explicite, une commande précise est requise pour lancer le calcul. Cela peut se faire en appelant une méthode de démarrage (`start`) ou dès la première utilisation de la valeur de la variable de type *future* ou *promesse*.

A.5 C++

En `C++`, la bibliothèque standard offre les variables de types *future* et *promise*, ainsi que le concept de «`async`» [36, 15, 16, 7, 6, 39] qui y est associé.

Les variables de type *future* servent à emmagasiner une valeur qui sera éventuellement disponible (comme un port de communication). Les variables de type `promise` servent à emmagasiner une valeur qui sera éventuellement transmise à une variable de type *future*. On associe cette dernière à une variable de type «`promise`» grâce à la commande «`get_future`». Ainsi tout ce qui est

assignée à la variable de type *promise* est reçu dans la variable de type *futur* associée. Toutes deux représentent les deux extrémités d'un port de communication unidirectionnel. Les seules actions permises sont alors un «`get`» sur la variable de type *future* (l'équivalent du `receive` sur un port) et un «`set_value`» sur une variable de type *promise* (l'équivalent du `send` sur un port). Les programmes A.6 et A.7 illustrent ces concepts. Le programme A.8 met en évidence l'usage des types «`future`» et «`promise`» avec des objets et des fonctions `lambdas`.

```cpp
#include <thread>
#include <future>
#include <iostream>

void fonction1(std::promise<int> * canal_s)
{
    std::cout<<"Exécution du fil (fonction1)"<<std::endl;
    canal_s->set_value(35);
}

int main()
{
    std::promise<int> canal_s;
    std::future<int> canal_r = canal_s.get_future();
    std::thread th(fonction1, &canal_s);
    std::cout<<canal_r.get()<<std::endl;
    th.join();
    return 0;
}
```

Programme A.6 – Exemple d'utilisation de variables de types «`future`» et «`promesse`» en `C++`

Selon la norme `C++` 2020 [39, 17, 5], les variables de type *future* fourniront de nouvelles extensions. Ainsi, réforme permet notamment à ce type de variable de :

- tester si elle est reliée à une variable de type *promise* (`valid`) ;
- tester si une valeur est disponible (`is_ready`) ;
- l'attacher à une autre variable de types *future* (`then`) ;
- initier plusieurs autres actions (`unwrap`, `when_any`, `when_all`, ...).

Le langage C++ fournit aussi un prototype de fonction, «`async`» [6, 15, 16], qui permet d'exécuter une fonction de façon asynchrone. Cette méta-fonction reçoit en paramètre une fonction, l'exécute généralement de façon asynchrone et retourne son résultat dans une variable de type *future*. En fait, l'énoncé «`async`» est une interface de haut niveau pour les fils d'exécution. Ainsi, l'environnement de `C++` (*run time*) maintient un groupe de fils d'exécution (thread pool) pour traiter les fonctions démarrées de façon asynchrone. L'unique situation pour laquelle une fonction démarrée par «`async`» ne pourrait être exécutée par un autre fil, serait celle où aucun fil n'est disponible dans le groupe. Les programmes A.9, A.10 et A.11 illustrent ces concepts.

```cpp
#include <thread>
#include <future>
#include <iostream>

void fonction1(std::future<int>*  canal_r)
{
    std::cout<<"Exécution du fil (fonction1) : ";
    int x = canal_r->get();
    std::cout << "réception de la valeur : " << x << std::endl;
}
int main()
{
    std::promise<int> canal_s;
    std::future<int> canal_r = canal_s.get_future();
    std::thread th(fonction1, &canal_r);
    canal_s.set_value(35);
    th.join();
    return 0;
}
```

Programme A.7 – Exemple d'utilisation de variables de types «`future`» et «`promesse`» en `C++`

A.6 Javascript

Le langage `Javascript` supporte seulement le concept de *promesses* qui, associées avec l'énoncé `async`, permet l'exécution de tâches asynchrones et la récupération des résultats. Les programmes A.12, A.13 et A.14 présentent des exemples d'application de ces concepts, tirés de différents documents d'introduction [11, 13, 9, 10]

A.7 Python

Dans le langage de programmation `Python`, seules les variables de type *future* existent et elles ressemblent beaucoup aux concepts de *promesse* de `Javascript`. Cependant, la prudence est de mise car dans les faits, deux types distincts de *future* se côtoient en `Python`. Le premier (`Concurrent.futures`) est associé aux groupes de fils d'exécution (Threadpool) ou de processus (Processpool), alors que le second est associé au module `Asyncio`. Ce dernier a été introduit depuis Python 3.5, puis modifié dans Python 3.6 et dans 3.7. À chaque version, le concept de «*future*» a subi de tels changements qu'il est important ici de s'attarder à bien définir ces concepts.

A.7.1 La bibliothèque «`concurrent.future`»

Le module «`concurrent.futures`» [21, 20, 26] se définit comme une abstraction permettant d'utiliser les fils d'exécution (`ThreadPoolExecutor`) ou les processus (`ProcessPoolExecutor`). Dans ces deux cas, un groupe de fils ou de processus est créé et géré par le module. Ce dernier assigne des tâches aux ressources disponibles dans le groupe.

À noter : tous les exemples cités dans cette section sont repris de [21, 20, 26].

Ce module de Python fournit donc des commandes pour initialiser un groupe de fils ou processus

```cpp
#include <future>
#include <iostream>
#include <thread>

void multiplier(std::promise<int> * canal, int a, int b)
{
  canal->set_value(a*b);
}
struct Diviser{
  void operator() (std::promise<int> * canal, int a, int b) const
  {
    canal->set_value(a/b);
  }
};

int main(){
  int a= 20;
  int b= 10;
  //definition du canal pour l'envoie
  std::promise<int> canal_mul;
  std::promise<int> canal_div;
  std::promise<int> canal_add;
  // definition du canal de reception
  std::future<int> resultat_mul = canal_mul.get_future();
  std::future<int> resultat_div = canal_div.get_future();
  std::future<int> resultat_add = canal_add.get_future();
  // calcul
  std::thread fil_mul(multiplier, &canal_mul, a, b);
  Diviser mon_div;
  std::thread fil_div(mon_div, &canal_div, a, b);
  std::thread fil_add([](std::promise<int> * canal, int a, int b)
                      {canal->set_value(a+b);}, &canal_add, a, b);
  // Obtention des résultats
  std::cout << "20*10= " << resultat_mul.get() << std::endl;
  std::cout << "20/10= " << resultat_div.get() << std::endl;
  std::cout << "20+10= " << resultat_add.get() << std::endl;
  // Attendre la fin des fil d'exécution
  fil_mul.join();
  fil_div.join();
  fil_add.join();
  std::cout << std::endl;
}
```

Programme A.8 – Exemple d'utilisation de variables de types «`future`» et «`promesse`» en `C++`

```cpp
#include <future>
#include <iostream>

void fonction1()
{
    std::cout << "Appel asynchrone" << std::endl;
}

int main() {
    //called_from_async launched in a separate thread if possible
    std::future<void> resultat( std::async(fonction1));

    std::cout << "Le programme principal s'exécute." << std::endl;
    resultat.get();
    return 0;
}
```

Programme A.9 – Exemple d'utilisation du «async» avec une fonction normale en C++

```cpp
#include <future>
#include <iostream>
using namespace std;
int main()
{
    future<int> resultat( std::async([](int m, int n)
            { cout << "Execution du fil" << endl;return m + n;} , 2, 4));
    cout << "Le programme principal s'exécute." << endl;
    cout << resultat.get() << endl;
    return 0;
}
```

Programme A.10 – Exemple d'utilisation du «async» avec une fonction lambda en C++

```cpp
#include <future>
#include <iostream>
#include <vector>
int carre(int c) {return c * c; }
int main() {
    std::vector<std::future<int>> resultats;
    for(int i = 0; i < 10; ++i) {
        resultats.push_back (std::async(carre, i));
    }
    for(auto &r : resultats) {
        std::cout << r.get() << std::endl;
    }
    return 0;
}
```

Programme A.11 – Exemple d'utilisation de multiples énoncés «async» en C++

```
let hello = async function() { return "Hello" };
hello();
------------------------------
let hello = async () => { return "Hello" };
------------------------------
hello().then((value) => console.log(value))
```

Programme A.12 – Exemple d'utilisation du «`async`» en `Javascript`

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then...

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Programme A.13 – Exemple d'utilisation des «`promesses`» en `Javascript`

```
promise.then(function(result) {
  console.log(result); // "Stuff worked!"
}, function(err) {
  console.log(err); // Error: "It broke"
});
```

Programme A.14 – Exemple d'utilisation des «`promesses`» en `Javascript`

(`ThreadPoolExecutor` ou `ProcessPoolExecutor`), démarrer des tâches (`submit` ou `map`), attendre la fin des tâches (`done`, `wait`, `as_completed`) et récupérer les résultats (*future* et `result`).

Le programme A.15 explicite la façon d'initialiser respectivement des groupes de fils et de processus en leur soumettant une tâche grâce à la commande «`submit`». La soumission d'une tâche retourne une variable de type *future* qui contiendra le résultat une fois la tâche achevée. Par la suite la fonction «`done`» retourne «`faux`» si le résultat n'est pas disponible. La fonction «`result`» retourne le résultat, s'il y a lieu.

```python
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def dort(msg):
    sleep(5)
    return msg

gr1 = ThreadPoolExecutor(3)

future = gr1.submit(dort, ("Bonjour"))

if not future.done():
       print("Attente")
       sleep(5)
print(future.result())
```

```python
from concurrent.futures import ProcessPoolExecutor
from time import sleep

def dort(msg):
    sleep(5)
    return msg

gr1 = ProcessPoolExecutor(3)

future = gr1.submit(dort, ("Bonjour"))

if not future.done():
       print("Attente")
       sleep(5)
print(future.result())
```

Programme A.15 – Exemples d'utilisation du module `concurrent.futures` en Python

Les programmes A.16, A.17 et A.18 exposent respectivement la procédure d'utilisation des fonctions «`as_completed`», «`wait`» et «`map`». Au programme A.16, la fonction «`as_completed`» permet de récupérer les résultats des fils aussitôt que ceux-ci terminent. Au programme A.17, la fonction «`wait`» permet d'attendre la fin du premier fil (`FIRST_COMPLETED`) pour récupérer ses valeurs de retour ou de tous les fils (`ALL_COMPLETED`). Au programme A.18, on initialise d'abord un groupe de fils, puis on recourt à «`map`» pour lancer l'exécution de toutes les fonctions en même temps. Dans le premier cas, le groupe contient cinq fils. Ainsi, après 3 secondes nous obtiendrons tous les résultats. Dans le second cas, le groupe ne contient que deux fils. Seulement deux programmes seront traités concurremment et, le tout prendra 9 secondes. Le module s'occupe de la planification sur les ressources disponibles. La fonction «`map`» retourne une liste de résultats.

Le programme A.19 illustre une autre utilisation de la fonction «`map`». Dans cet exemple, la fonction `zip` sert à regrouper deux listes de même longueur en une liste de paires de valeurs.

Finalement, le programme A.20 indique comment spécifier une fonction de continuation (appelée `add_done_callback`) pour une tâche qui s'exécute dans un groupe de fils.

A.7.2 Async/await et future

La seconde implantation du type «`future`» se retrouve dans le module «`asyncio`» [21, 20, 26, 4, 8, 14, 24, 38, 37, 43, 12, 42, 22, 29, 28, 30, 3, 33, 31] et est principalement basée sur les coroutines de Python.

L'usage des coroutines a beaucoup évolué de Python 3.3 à 3.7. Abordons la syntaxe du `asyncio` version 3.6 (version que j'utilise présentement). Celle-ci est différente de la syntaxe de 3.3 et a changé à nouveau dans 3.7. On commentera brièvement les dernières versions (3.7 et plus) mais sans toutefois pouvoir en tester le code.

```python
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

from time import sleep
from random import randint

def attendre(no):
    sleep(randint(1, 5))
    return "Fin du fil no.{}".format(no)

gr1 = ThreadPoolExecutor(5)

futures = []

for id in range(5):
    futures.append(pool.submit(attendre, id))


for fil in as_completed(futures):
    print(fil.result())
```

```python
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed
import urllib.request

URLS = ['https://www.usherbrooke.ca/',
        'https://www.usherbrooke.ca/sciences/',
        'https://www.usherbrooke.ca/info/',
        'https://www.ledevoir.com/',
        'https://www.latribune.ca/']

def charger(url):
  with urllib.request.urlopen(url) as lien:
      return lien.read()

with ThreadPoolExecutor(max_workers=5) as ex:
  res = {ex.submit(charger, url):
      url for url in URLS}
  for fut_res in as_completed(res):
     adr = res[fut_res]
     page = fut_res.result()
     print('%r contient %d oct.' % (adr,len(page)))
```

Programme A.16 – Exemples d'utilisation du «`as_completed`» (`concurrent.futures`)

```python
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait
from concurrent.futures import FIRST_COMPLETED

import random
import itertools
import time

def divise(n):
   t = random.randint(1, 5)
   time.sleep(t)
   return n / 10

with ThreadPoolExecutor() as gr1:
  fut = {gr1.submit(divise, ident) :
     ident for ident in range(1,6,1)}
  i = 1

  while fut:
     term, fut = wait(fut, return_when=FIRST_COMPLETED
     for res in term :
        val = res.result()
        print(f"{i} : fil {int(val*10)} : res= {val}")
        i=i+1
```

```python
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait
from concurrent.futures import ALL_COMPLETED
from concurrent.futures import FIRST_COMPLETED
import random, time, itertools

def divise(n):
    t = random.randint(1, 5)
    time.sleep(t)
    return n / 10

with ThreadPoolExecutor() as gr1:
    fut = {gr1.submit(divise, ident) :
        ident for ident in range(1,6,1)}
    i = 2
    term, fut = wait(fut, return_when=FIRST_COMPLETED)

    for t in term: val = t.result()
    print(f"PREMIER : fil {int(val*10)} : res= {val}")
    term, fut = wait(fut, return_when=ALL_COMPLETED)
    for res in term :
        val = res.result()
        print(f"{i} : fil {int(val*10)} : res= {val}")
        i=i+1
```

Programme A.17 – Exemples d'utilisation du «`wait`» de `concurrent.futures`

```python
from concurrent import futures
import time

def divise(n):
    time.sleep(3)
    return n / 10

ex = futures.ThreadPoolExecutor(5)

resultats = ex.map(divise, range(5, 0, -1))
for i in resultats : print (i)
```

```python
from concurrent import futures
import time

def divise(n):
    time.sleep(3)
    return n / 10

ex = futures.ThreadPoolExecutor(2)

resultats = ex.map(divise, range(5, 0, -1))
for i in resultats : print (i)
```

Programme A.18 – Exemples d'utilisation du «map» de `concurrent.futures`

```python
from concurrent.futures import ProcessPoolExecutor
import math
PREMS = [
    112272535095293,
    112582705942171,
    112272535095293,
    115280095190773,
    115797848077099,
    1099726899285419]

def est_prem(nb):
    if nb % 2 == 0: return False
    racine = int(math.floor(math.sqrt(nb)))
    for i in range(3, racine + 1, 2):
        if nb % i == 0: return False
    return True

def main():
   with ProcessPoolExecutor() as gr:
      res = gr.map(est_prem, PREMS)
      for nb, sont_prem in zip(PREMS, res):
          print('%d est premier? %s' %(nb, sont_prem))

if __name__ == '__main__': main()
```

Programme A.19 – Exemples d'utilisation du «map» de `concurrent.futures`

```
from concurrent.futures import ThreadPoolExecutor
import time
def divise(n):
    time.sleep(0.5)
    return n / 10

def fin(fn):
    if fn.done():
        result = fn.result()
        print('{}: fil res= {}\n'.format(
               fn.arg, result))

if __name__ == '__main__':
    ex = ThreadPoolExecutor(max_workers=2)
    print('DEBUT..........')
    f = ex.submit(divise, 5)
    f.arg = 666
    f.add_done_callback(fin)
    res = f.result()
    print(".....",res)
```

Programme A.20 – Exemples d'utilisation du «`map`» de `concurrent.futures`

Le coroutines de Python (`asyncio`) font aussi appel à la boucle d'événements, soit une boucle qui attend que des événements se produisent pour lancer certaines tâches.

Coroutines

Les coroutines de Python sont légèrement différentes de celles que nous avons déjà définies. Elles s'assimilent en fait aux fils d'exécution de niveau usager. Ainsi, les coroutines de Python sont basée sur le multi-tâches coopératif, ce qui implique que chacune des coroutines relâche volontairement le contrôle de l'UCT. En fait, dans le cas contraire, elle monopoliserait l'UCT et les autres coroutines ne s'exécuteraient jamais. Toutefois, les coroutines ne déterminent pas par elles-mêmes la prochaine coroutine à prendre le contrôle. L'environnement «`asyncio`» fournit un planificateur qui a cette charge. C'est la boucle d'événements.

Pour utiliser les coroutines il faut donc :

- définir une coroutine (`async def`) ;
- planifier son exécution (la mettre en file d'attente), soit en lançant directement son exécution avec la commande «`await`», soit en créant une variable de type *future* ou une tâche.
- lancer son exécution. Cela consiste à donner le contrôle au planificateur de la boucle d'événements.

Définition de coroutines

La déclaration d'une coroutine est très similaire à celle d'une fonction en Python. Pour créer une coroutine, il suffit de recourir au mot réservé «`async def`» (au lieu de `def` pour une fonction normale), comme l'illustre le programme A.21. Notons que le nombre de coroutines n'est pas limité, on en définit autant que nécessaire pour répondre à nos besoins.

Planifier et lancer l'exécution d'une coroutine

```python
import asyncio

async def coro1():
    print('Bonjour de la coroutine')
```

Programme A.21 – Exemple de création d'une coroutine

Même si la déclaration d'une coroutine est similaire à celle d'une fonction, il n'en reste pas moins que ce n'est pas une fonction normale. Il est important de comprendre qu'un appel «standard» à une coroutine ne démarre ni ne planifie l'exécution de celle-ci. Pour en arriver à cette étape, il faut d'abord créer une boucle d'événements qui, elle, lancera l'exécution de la coroutine. Le programme A.22 indique le procédé adéquat.

On y remarque que la charge de planifier l'exécution de la coroutine et de transférer le contrôle à la boucle d'événements (démarre l'exécution de la coroutine) revient au programme principal. La commande «`asyncio.get_event_loop()`» sert à identifier la boucle d'événements courante. La commande «`loop.run_until_complete(coro1())`» permet au programme principal de se mettre en attente jusqu'à ce que la coroutine («coro1» dans ce cas), ou la tâche spécifiée en paramètre, termine son exécution. Cette même commande sert à planifier et à lancer l'exécution de la coroutine. À partir de Python 3.7, ces deux commandes ont été remplacées par «`asyncio.run`».

```python
# Python 3.5
import asyncio
async def coro1():
    print('Bonjour de la coroutine')

loop = asyncio.get_event_loop()
loop.run_until_complete(coro1())
loop.close()
```

```python
# Python 3.7
import asyncio
async def coro1():
    print('Bonjour de la coroutine')



asyncio.run(coro1())
```

Programme A.22 – Exemple de création et d'exécution d'une coroutine

Planifier et lancer l'exécution de plusieurs coroutines

L'exécution de plusieurs sous-routines passent d'bord par leur planification. Elle se fait directement à même une autre coroutine, ou en créant une tâche ou une «*future*». La création d'une tâche se fait soit statiquement (avant le démarrage de la boucle), soit dynamiquement (pendant l'exécution de la boucle). Une fois qu'elles sont ainsi planifiées, la boucle d'événements se charge de les démarrer une par une (elle fait la répartition). Les différents modes de planification sont :

- la commande «`await`»

  Cette commande permet à une coroutine de lancer immédiatement l'exécution d'une autre coroutine et d'attendre la fin de son exécution. Cette commande n'est utilisable qu'à l'intérieur d'une coroutine. Le programme A.23 illustre l'emploi de la commande «`await`».

- La commande «`asyncio.gather`»

  Cette commande permet de démarrer plusieurs coroutines simultanément. Lorsque leur exécution prend fin, la commande organise tous les résultats retournés par les coroutines. Cette

```
import asyncio
async def coro1():
    print('Début de la coroutine 1')
    await coro2()
    print('Fin de la coroutine 1')

async def coro2():
    print('Début de la coroutine 2')
    await coro3()
    print('Fin de la coroutine 2')

async def coro3():
    print('Exécution de la coroutine')

loop = asyncio.get_event_loop()
loop.run_until_complete(coro1())
loop.close()
```

Programme A.23 – Exemple de création et d'exécution de plusieurs coroutines

commande est employée pour lancer la boucle d'événements ou pour démarrer des coroutines à partir d'une coroutine. Le programme A.24 présente un exemple d'utilisation de cette commande.

Dans les futures versions de Python (3.10), cette commande ne sera plus disponible.

- Les commandes «`asyncio.create_task`», «`asyncio.ensure_future`» et «`loop.create_task`»

  Ces commandes sont des plus similaires. Elles permettent à une coroutine ou au programme principal, et ce avant le démarrage de la boucle d'événements, de planifier l'exécution d'une ou plusieurs coroutines sans attendre la fin de leur exécution. La commande «`asyncio.create_task`» est valide seulement à partir de Python 3.7 et est destinée à remplacer «`asyncio.ensure_future`». Ultimement les deux premières fonctions font appel à la fonction «`loop.create_task`».

  Les programmes A.25 et A.26 sont des exemples de planification statique de coroutines. Il est aussi possible d'ajouter dynamiquement des coroutines grâce à ces mêmes commandes. Le programme A.27 illustre ce cas.

  Il est important de noter qu'une variable de type *future* est un objet spécial qui représente toujours un résultat éventuel. En fait une variable de ce type est une tâche munie de fonctionnalités additionnelles (une tâche est une sous-classe de la classe *future*). Dans la bibliothèque «`asyncio`», on ne devrait guère recourir aux objets de type *future*. On les consacre plutôt à des besoins de communication ou de continuation (callback).

- «`asyncio.wait`»

  Cette commande permet au programme principal d'exécuter une coroutine et d'attendre la fin de son exécution.

Communication et fonction de rappel (callback)

```python
import asyncio

async def co1():
    await asyncio.sleep(3)
    print('Bonjour de la coroutine 1')

async def co2():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 3')


loop = asyncio.get_event_loop()
taches = asyncio.gather(co1(), co2(), co3())
loop.run_until_complete(taches)
loop.close()
```

```python
import asyncio
async def co1():
    await asyncio.sleep(3)
    print('Bonjour de la coroutine 1')

async def co2():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 3')

async def main():
    await asyncio.gather(co1(), co2(), co3())

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Programme A.24 – Exemple de création et d'exécution de coroutines

```python
import asyncio
async def co1():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 1')
    await asyncio.sleep(5)

async def co2():
    await asyncio.sleep(1)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(5)
    print('Bonjour de la coroutine 3')

loop = asyncio.get_event_loop()

loop.create_task(co1())
loop.create_task(co2())
loop.create_task(co3())

loop.run_until_complete(co1())
loop.close()
```

Programme A.25 – Exemple de création et d'exécution d'une coroutine

```python
import asyncio
async def co1():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 1')

async def co2():
    await asyncio.sleep(1)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(5)
    print('Bonjour de la coroutine 3')



aynsio.create_task(co1())
aynsio.create_task(co2())
aynsio.create_task(co3())

loop = asyncio.get_event_loop()
loop.run_until_complete(co1())
loop.run_until_complete(co3())
loop.close()
```

```python
import asyncio
async def coro1():
    print('Début de la coroutine 1')
    await asyncio.sleep(1)
    print('Fin de la coroutine 1')

async def coro2():
    print('Début de la coroutine 2')
    await asyncio.sleep(2)
    print('Fin de la coroutine 2')

async def coro3():
    print('Exécution de la coroutine 3')
    await asyncio.sleep(3)
    print('Fin de la coroutine 3')

t1 = asyncio.ensure_future(coro1())
t2 = asyncio.ensure_future(coro2())
t3 = asyncio.ensure_future(coro3())

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(t1,t2,t3))
loop.close()
```

Programme A.26 – Exemple de création et d'exécution de coroutines

```python
import asyncio
async def co1():
    await asyncio.sleep(3)
    print('Bonjour de la coroutine 1')

async def co2():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 3')

async def main(boucle):
    boucle.create_task(co1())
    boucle.create_task(co2())
    boucle.create_task(co3())
    await asyncio.sleep(5)
    print('Bonjour de main')

loop = asyncio.get_event_loop()
loop.run_until_complete(main(loop))
loop.close()
```

```python
import asyncio
async def co1():
    await asyncio.sleep(3)
    print('Bonjour de la coroutine 1')

async def co2():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 2')

async def co3():
    await asyncio.sleep(2)
    print('Bonjour de la coroutine 3')

async def main():
    asyncio.ensure_future(co1())
    asyncio.ensure_future(co2())
    asyncio.ensure_future(co3())
    await asyncio.sleep(5)
    print('Bonjour de main')

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

Programme A.27 – Exemple de création et d'exécution de coroutines

Lorsque l'on utilise une coroutine, il est possible de retourner des valeurs grâce à la commande «`return`» qui agit comme celle d'une fonction normale. Le programme A.28 propose un transfert de valeurs à l'aide de la commande «`return`» et l'emploi du «`gather`» pour récupérer de nombreuses valeurs.

```
import asyncio

async def f(x):
    y = await g(x)
    return y * 4

async def g(x):
   return x+5

boucle = asyncio.get_event_loop()

res = boucle.run_until_complete(f(10))
boucle.close()


print(res)
```

```
import asyncio
from pprint import pprint
async def f(x):
    return x * 4

async def g(x):
   return x+5

async def h(x):
   return x-3

boucle = asyncio.get_event_loop()
futures = asyncio.gather(f(10),g(35),h(13))
res = boucle.run_until_complete(futures)
boucle.close()
for r in res :
   print(r)
```

Programme A.28 – Exemple d'utilisation de la commande `return`

L'objet de type *future* de «`asyncio`» nous offre la capacité de faire communiquer plusieurs coroutines et de planifier des fonctions de rappel. Le programme A.29 illustre ces faits.

Quelques remarques sur «`Asyncio`» et les «`Futures`»

Il s'avère important de spécifier quelques points :

- Une tâche est une sous-classe d'une *future*.
- Appeler du code asynchrone (`async`) code à partir de code **asynchrone** implique le recours au «`await`» ou à une variable de type *future* pour obtenir le résultat.
- Appeler du code asynchrone (`async`) code à partir de code **synchrone** implique la création d'une boucle d'événements et l'ajout explicite du code asynchrone dans la boucle.
- Il est possible de démarrer une boucle d'événements dans un autre fil avec les commandes suivantes :
  - exec = concurrent.future.threadPoolExecutor ;
  - loop = asyncio.geteventloop ;
  - result = await loop.run_in_executor (exec, fonction, param).
- Le premier appel aux coroutines donne le contrôle à la boucle d'événements qui, elle, planifie les tâches en attente.
- La différence entre les coroutines et les fils réside dans le fait qu'il n'y a pas de réquisition avec les coroutines. Elles doivent relâcher le contrôle volontairement (multi-tâche coopératif). Les fils sont sujets à des interruptions à tout moment.

```python
import asyncio
async def produc(fut1, fut2):

        print("Production 1")
        await asyncio.sleep(1)
        fut1.set_result('terminé 1')
        await asyncio.sleep(1)
        fut2.set_result('terminé 2')

async def consom(fut):
       print('Attente du résultat ')
       res = await fut
       print('Le résultat:{!r}'.format(res))

boucle = asyncio.get_event_loop()

fut1 = asyncio.Future()
fut2 = asyncio.Future()

prod = boucle.create_task(produc(fut1,fut2))
cons1 = boucle.create_task(consom(fut1))
cons2 = boucle.create_task(consom(fut2))

boucle.run_until_complete(cons2)
boucle.close()
```

```python
import asyncio
async def fonction1():
    await asyncio.sleep(5)
    return 'Bonjour de la fonction 1!'
async def fonction2():
    await asyncio.sleep(2)
    return 'Bonjour de la fonction 2!'

def recup1(future):
    print(future.result())
def recup2(future) :
    print(future.result())

async def bonjour():
    fut1 = asyncio.ensure_future(fonction1())
    fut2 = asyncio.ensure_future(fonction2())
    fut1.add_done_callback(recup1)
    fut2.add_done_callback(recup2)
    await asyncio.sleep(3)
    print("---------------------------------")
    await asyncio.sleep(5)

boucle = asyncio.get_event_loop()
res = boucle.run_until_complete(bonjour())
boucle.close()
```

Programme A.29 – Exemple d'utilisation du `return` pour faire communiquer des coroutines

- Une coroutine s'exécute de façon synchrone jusqu'à ce qu'elle rencontre un «`await`». Elle se met alors en attente et le contrôle est transféré à une autre coroutine par la boucle d'événements.
- Il est recommandé de n'utiliser que les commandes «`asyncio.create_task`» et «`loop.create_task`» pour démarrer des coroutines car l'emploi de la commande «`ensure_future`» est généralement réservé à d'autres types de tâches.
- La commande «`ensure_future`» s'assure que l'objet passé en paramètre soit toujours transformé en variable de type *future*. Ainsi, si cette fonction reçoit :
  - une coroutine, elle retourne une valeur de type *future* ;
  - une *future*, elle retourne une valeur de type *future* ;
  - une tâche, elle retourne une tâche (qui est une *future*).

  En fait si elle reçoit une *future* ou une tâche, elle ne prend aucune action.
- La commande «`run_until_complete`» démarre une seule coroutine qui, elle, a la charge de démarrer les autres.
- Généralement, on recommande d'employer les *futures* pour la continuation ou la communication.
- La fonction «`await`» est un peu l'équivalent du «`yield`» (fil d'exécution) ou du «`resume`» (coroutine telle que déjà définie en classe).

Autres fonctionnalités

La bibliothèque «`asyncio`» contient plusieurs autres fonctionnalités que nous ne présenterons pas ici. Pour plus de détails, veuillez consulter les différents documents d'introduction disponibles sur le Web [21, 20, 26, 4, 8, 14, 24, 38, 37, 43, 12, 42, 22, 29, 28, 27, 30, 3, 33, 31].

A.8 Rust

À venir...

A.9 Pour plus d'informations

Plusieurs autres caractéristiques sont associées aux *futurs* et aux *promesses*. En premier lieu, il y a le «pipelining» qui permet de réduire encore plus la latence des communications en reliant entre elles des promesses. Il y a aussi des sémantiques bloquantes ou non, l'évaluation paresseuse, etc.

Vous trouverez toutes les informations sur ces caractéristiques dans [25, 40]

Bibliographie

[1] Adrian ANCONA : Futures, async, packaged_tasks and promises in c++. https://ncona.com/2018/02/futures-async-packaged_tasks-and-promises-in-c/, 2018.

[2] Jake ARCHIBALD : Javascript promises : An introduction. https://web.dev/promises/, 2013.

[3] Brett CANNON : How the heck does async/await work in python 3.5 ? https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/, 2016.

[4] Alex CHAN : Adventures in python with concurrent.futures. https://alexwlchan.net/2019/10/adventures-with-concurrent-futures/, 2019.

[5] The C++ Standards COMMITTEE : std : :future. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3721.pdf, 2013.

[6] CPLUSPLUS.COM : std : :async. http://www.cplusplus.com/reference/future/async/?kw=async, 2020.

[7] CPLUSPLUS.COM : std : :future. http://www.cplusplus.com/reference/future/future/, 2020.

[8] Loris CRO : Async/await programming basics with python examples. https://redislabs.com/blog/async-await-programming-basics-python-examples/, 2019.

[9] MDN Web DOC : async function. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/async_function, 2019.

[10] MDN Web DOC : Graceful asynchronous programming with promises. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises, 2019.

[11] MDN Web DOC : Promise. https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Promise, 2019.

[12] EDUCATIVE : Python concurrency : Making sense of asyncio. https://www.educative.io/blog/python-concurrency-making-sense-of-asyncio, 2019.

[13] Eric ELLIOTT : Master the javascript interview : What is a promise ? https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-promise-27fc71e77261, 2017.

[14] Andrew Godwin : Python & async simplified. https://www.aeracode.org/2018/02/19/python-async-simplified/, 2018.

[15] Rainer Grimm : Promise and future. https://www.modernescpp.com/index.php/promise-and-future, 2016.

[16] Rainer Grimm : Tasks. https://www.modernescpp.com/index.php/tasks, 2016.

[17] Rainer Grimm : std : :future extensions. https://www.modernescpp.com/index.php/std-future-extensions, 2017.

[18] Per Brinch Hansen : The nucleus of a multiprogramming system. *Commun. ACM*, 13(4): 238–241, avril 1970.

[19] Per Brinch Hansen : *RC 4000 Software : Multiprogramming System*, page 153–197. Springer-Verlag, Berlin, Heidelberg, 2002.

[20] The Python Standard Library : concurrent.futures — launching parallel tasks. https://docs.python.org/3/library/concurrent.futures.html, 2020.

[21] Abu Ashraf Masnun : Python : A quick introduction to the concurrent.futures module. http://masnun.com/2016/03/29/python-a-quick-introduction-to-the-concurrent-futures-module.html, 2017.

[22] Mike : Python 3 – an intro to asyncio. https://www.blog.pythonlibrary.org/2016/07/26/python-3-an-intro-to-asyncio/, 2016.

[23] James L. Peterson et Abraham. Silberschatz : *Operating system concepts / James L. Peterson, Abraham Silberschatz.* Addison-Wesley Pub. Co Reading, Mass, 1983.

[24] Dan Poirier : Asyncio. https://cheat.readthedocs.io/en/latest/python/asyncio.html, 2019.

[25] Kisalaya Prasad, Avanti Patil et Heather Miller : Futures and promises. dist-prog-book.com/chapter/2/futures.html, 2016.

[26] PyMOTW-3 : concurrent.futures — manage pools of concurrent tasks. https://pymotw.com/3/concurrent.futures/, 2018.

[27] Python : asyncio — entrées/sorties asynchrones. https://docs.python.org/fr/3/library/asyncio.html, 2020.

[28] Python : Coroutines and tasks (3.8). https://docs.python.org/3/library/asyncio-task.html, 2020.

[29] Python : Coroutines et tâches (3.7). https://docs.python.org/fr/3.7/library/asyncio-task.html, 2020.

[30] Python : Tasks and coroutines (3.6). https://docs.python.org/3.6/library/asyncio-task.html, 2020.

[31] Python : Tâches et coroutines (3.7). https://docs.python.org/fr/3.6/library/asyncio-task.html, 2020.

[32] Spencer Rathbun : Parallel processing with promises. *Queue*, 13(3):10 :10–10 :18, mars 2015.

[33] Scott Robinson : Python async/await tutorial. https://stackabuse.com/python-async-await-tutorial/, 2020.

[34] Abraham Silberschatz, Greg Gagne et Peter Baer Galvin : *Operating System Concepts*. Wiley, 6 édition, 2002.

[35] Abraham Silberschatz, Peter Baer Galvin et Greg Gagne : *Operating System Concepts, 10th Edition*. Wiley, 2018.

[36] Paul Silisteanu : C++11 async tutorial. https://solarianprogrammer.com/2012/10/17/cpp-11-async-tutorial/, 2012.

[37] Brad Solomon : Async io in python : A complete walkthrough. https://realpython.com/async-io-python/, 2018.

[38] Victor Stinner : Asyncio documentation. https://asyncio.readthedocs.io/en/latest/, 2016.

[39] Varun : C++11 multithreading - part 9 : std : :async tutorial & example. https://thispointer.com/c11-multithreading-part-9-stdasync-tutorial-example/, 2017.

[40] Wikipedia : Futuresand promises. https://en.wikipedia.org/wiki/Futures_and_promises, 2019.

[41] Wikipedia : Rc 4000 multiprogramming system. https://en.wikipedia.org/wiki/RC_4000_multiprogramming_system, 2020.

[42] Peter Xi : Asyncio is not parallelism. https://towardsdatascience.com/asyncio-is-not-parallelism-70bfed470489, 2020.

[43] Serdar Yegulalp : 3 steps to a python async overhaul. https://www.infoworld.com/article/3562577/3-steps-to-a-python-async-overhaul.html, 2020.