GABRIEL LAVOIE

Programmation distribuée et migration de processus

Mémoire présenté
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de maîtrise en informatique
pour l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2011

Résumé

Ce mémoire propose un modèle de programmation distribuée basé sur la migration de processus pouvant communiquer à l'aide de canaux de communication. Ce travail cadre bien avec le contexte actuel où l'augmentation de la puissance de traitement passe par les processeurs multicœurs et les systèmes distribués, qui permettent l'exécution de plusieurs processus en parallèle. L'étude de différentes algèbres de processus et langages de programmation permettant le parallélisme a tout d'abord permis de comparer leurs différentes caractéristiques. Suite à cette étude, nous présentons différents concepts nécessaires à la mise en place de notre modèle distribué par migration de processus, dans le cadre des langages objets qui imposent certaines contraintes. Finalement, l'implémentation de notre modèle à l'aide des fonctionnalités de Stackless Python permet de voir comment chacun des concepts a été mis en place. Cette implémentation se présente sous la forme d'une extension qui permet la transformation de programmes existants en programmes distribués.

Table des matières

Table des figures

Liste des tableaux

Introduction

En jetant un coup d'œil à la fabrication des processeurs, on se rend rapidement compte que la course au gigahertz est terminée depuis plusieurs années. Au cours de ces années, le marketing des grands fabricants a instillé très profondément dans le marché l'idée qu'il fallait augmenter la fréquence des processeurs pour augmenter leur performance. En conséquence, dès qu'on envisageait l'achat d'un processeur, on considérait systématiquement celui ayant la plus haute fréquence. Par ailleurs, lorsqu'il a mis sur le marché sa plate-forme *Pentium 4*, le fabriquant Intel visait une fréquence de 10 GHz. Il n'a jamais réussi à atteindre cette fréquence, car les problèmes de surchauffe et de fuites électriques dans les semi-conducteurs sont devenus insurmontables.

Pendant cette même période, en produisant des processeurs plus performants que ceux d'Intel et fonctionnant à plus basse fréquence, le fabriquant AMD a démontré que l'augmentation de la fréquence n'était pas la solution. Il s'est par contre buté à la croyance fortement ancrée dans le marché que la vitesse équivalait à la performance. Il dut alors nommer ses processeurs avec des numéros qui ne correspondaient pas à des GHz, mais qui étaient à peu près équivalents aux valeurs en GHz qu'Intel utilisait pour ses processeurs. Intel a fini par suivre AMD et n'utilise maintenant plus la fréquence pour mesurer la performance de ses processeurs.

Conséquemment, les fabricants utilisent désormais des processeurs à plusieurs cœurs pour augmenter la performance des ordinateurs. Ils peuvent être vus comme l'assemblage de plusieurs processeurs dans un seul boîtier. Chacun des cœurs est indépendant et ensemble, ils peuvent effectuer différents traitements en même temps. Les systèmes comportant plusieurs processeurs ne sont pas nouveaux, mais ils étaient réservés jusqu'à tout récemment aux serveurs et à certaines applications spécialisées. L'utilisation de processeurs multicœurs est avantageuse et permet de réduire les coûts. Il n'est maintenant plus nécessaire d'utiliser des cartes mères spécialisées permettant d'accueillir plus d'un processeur, bien qu'elles existent toujours dans le but de recevoir plusieurs processeurs multicœurs.

Problématique

Pour tirer profit d'un système comportant plusieurs processeurs à un cœur ou bien un processeur multicœur, les programmes informatiques doivent être adaptés. En effet, un programme composé d'un seul fil d'exécution et qui traite beaucoup d'information ne tirera profit que d'un seul cœur, car les instructions sont exécutées les unes à la suite des autres. Le programmeur doit identifier lui-même les différentes parties d'un traitement qui peuvent être exécutées de façon parallèle. Chacune de ces parties doit être implémentée à l'aide de processus distincts qui communiquent à l'aide de canaux ou de plusieurs fils d'exécution qui partagent un même espace mémoire. Le système d'exploitation est alors en mesure de distribuer chacun des processus sur l'ensemble des cœurs à sa disposition ce qui permet d'accélérer le traitement.

Tous ces mécanismes sont offerts au programmeur par le système d'exploitation et ne sont pas simples à mettre en place. De plus, plusieurs problèmes peuvent survenir. Par exemple, avec l'utilisation de fils d'exécution sur une mémoire partagée, des variables peuvent être modifiées par un fil avant qu'un autre fil n'ait eu le temps de traiter la valeur qu'il venait de lui attribuer. Ce phénomène se produit, peu importe que les fils soient exécutés sur un seul ou sur plusieurs cœurs, car les changements de contexte nécessaires à l'exécution de plusieurs fils d'exécution peuvent survenir à tout moment.

Pour remédier à ce problème, les systèmes d'exploitation offrent des mécanismes pour bloquer l'accès simultané à certaines données ou à certains bouts d'un programme. Par contre, ces mécanismes sont souvent très difficiles à utiliser. Par exemple, le programmeur peut facilement créer par erreur deux fils d'exécution qui restent bloqués sur l'accès à une zone mémoire commune, car un troisième fil ne libère jamais le verrou qu'il a posé sur cette même zone mémoire après l'avoir utilisée. Ce genre de problème est difficile à diagnostiquer et à régler, car la seule chose que le programmeur voit est un blocage d'une partie ou de l'ensemble du programme. De plus, ces blocages surviennent souvent de façon aléatoire et sont difficiles à reproduire.

L'utilisation de canaux de communication aide à régler ce genre de problème, car il n'y a plus de mémoire partagée entre les processus. Les programmes écoutent sur des canaux lorsqu'ils ont besoin d'une certaine information, jusqu'à ce qu'un autre processus fournisse celle-ci. Les communications par canaux peuvent se faire de manière synchrone ou asynchrone. Dans la manière synchrone, un processus qui envoie une donnée sur un canal reste bloqué tant et aussi longtemps que cette donnée n'est pas lue par un autre processus. Dans la manière asynchrone, la donnée envoyée sur un canal est ajoutée à une file d'attente de traitement et le processus émetteur continue son exécution, peu

importe que le processus receveur ait lu ou non la donnée. Les blocages sont toujours possibles, mais il est habituellement plus simple de diagnostiquer une écriture qui n'a pas été faite sur un canal qu'un blocage de plusieurs fils d'exécution à cause d'un verrou non libéré.

La programmation parallèle est aussi très utilisée dans des contextes de fermes de traitement. Ces fermes sont constituées de plusieurs serveurs reliés par un réseau à très haut débit. Ces serveurs traitent de façon parallèle et avec une puissance totale très grande des données qui sont reliées entre elles. Un exemple d'utilisation de fermes de serveurs est l'encodage vidéo de films d'animation. Il existe des outils pour échanger facilement des données entre les différents serveurs par des canaux de communication, mais les applications destinées à être exécutées sur ces fermes de serveurs sont encore très spécialisées et le programmeur doit tenir compte des systèmes sur lesquels ces applications seront exécutées. La mise en place de ces systèmes est souvent très laborieuse, car le logiciel distribué qui sera utilisé doit être installé sur chacun des serveurs utilisés pour les traitements.

Compte tenu de l'existence des processeurs multicœurs et des fermes de serveurs permettant de traiter des données en parallèle, compte tenu également des différences entre les outils permettant d'exploiter chacun de ces modèles de parallélisme, il serait intéressant de concevoir un modèle unifié permettant d'abstraire ces différences. On peut facilement considérer plusieurs processeurs sur un même ordinateur ou plusieurs ordinateurs reliés par un réseau comme un concept générique d'unités de traitement parallèles. Cette vision soulève plusieurs questions concernant les manières de gérer les données à traiter et la communication entre les différentes unités de traitement. Ce mémoire présente un tel modèle de parallélisme et propose des réponses à certaines de ces questions.

Objectifs et méthodologie

L'objectif premier de ce projet de recherche était le développement d'un outil de programmation parallèle et distribuée basé sur la migration de processus. Un tel modèle permet le déplacement de certaines parties d'un programme vers un autre ordinateur, pendant qu'elles sont en cours d'exécution.

Tout d'abord, il a été nécessaire de faire une étude de quelques algèbres de processus, langages de programmation et extensions de langages de programmation existants, permettant l'exécution de processus de façon parallèle. L'objectif de cette étude était

de bien comprendre la façon dont le parallélisme est présenté d'un point de vue algébrique, mais aussi la façon dont différents langages de programmation et extensions de langages l'implémentent. Plusieurs caractéristiques de chacun d'entre eux, reliées au parallélisme, ont été rassemblées.

À partir de l'étude des algèbres et langages, un ensemble de concepts a été retenu dans l'objectif de définir un modèle de programmation qui abstrait le plus possible les différences entre la programmation parallèle et la programmation distribuée. Ce modèle est basé sur la migration de processus utilisant des canaux de communication. Les différents concepts sont explorés dans le cadre des langages-objets qui imposent plusieurs contraintes par rapport à la façon dont différents processus peuvent communiquer ; contraintes qui, pour la plupart, n'existent pas dans les langages fonctionnels. Chacune de ces contraintes a été analysée et nous proposons des méthodes et des outils permettant de les éviter.

Plan

Dans le but de bien couvrir la manière dont les objectifs ont été atteints, ce mémoire est structuré de la façon suivante :

- Le premier chapitre présente trois algèbres de processus et cinq langages de programmation ou extensions de langages, permettant la mise en parallèle de processus. Les principales caractéristiques de chacun d'entre eux sont décrites et quelques exemples sont présentés. À la fin du chapitre, des tableaux permettent de faire une comparaison rapide de chacune des caractéristiques ayant été vues.
- Le deuxième chapitre présente les différents concepts nécessaires à la mise en place d'outils de programmations distribués, basés sur la migration de processus. Ces outils sont présentés dans le cadre des langages objets qui imposent plusieurs contraintes. Les différentes méthodes et outils permettant d'éviter des contraintes sont expliqués.
- Le troisième chapitre couvre l'implémentation du modèle de programmation distribuée à l'aide de Stackless Python, sous la forme d'un module nommé *dstackless*. Les concepts présentés précédemment sont repassés un à un, par rapport à leur mise en place avec Stackless Python et ses caractéristiques pouvant être exploitées pour simplifier cette implémentation sont expliquées à l'aide d'exemples.
- Dans le quatrième chapitre, quelques applications concrètes sont démontrées à l'aide du modèle de programmation présenté dans ce mémoire. On présente tout d'abord l'historique de chacune de ces applications ainsi les outils actuels. Par la suite, des exemples de leurs mises en place sont présentés à l'aide de *dstackless*.

Chapitre 1

Algèbres de processus et langages concurrents

Ce chapitre est composé de deux parties :

1. La première partie est consacrée à la présentation de différentes algèbres de processus utilisées abondamment dans la littérature. Plus précisément, on présente les algèbres suivantes : CCS, π-calcul et le calcul ambiant.
2. La deuxième partie est consacrée à la présentation de différents langages de programmation, ou différentes extensions (généralement sous forme de paquetages) de langages existants, permettant la programmation de processus s'exécutant en parallèle et interagissant à travers des canaux de communication.

À la fin du chapitre, on présente un tableau comparatif permettant de distinguer les différentes caractéristiques des différents langages ou extensions présentés.

1.1 Algèbres de processus

Les algèbres de processus se révèlent utiles pour l'étude et la compréhension des aspects concurrent, parallèle et distribué des langages de programmation. Plus précisément, une algèbre de processus est un langage permettant la description formelle de systèmes complexes, plus particulièrement, ceux qui s'exécutent et qui communiquent de manière parallèle.

Dans la littérature, nous retrouvons une panoplie d'algèbres définies pour la des-

cription et l'étude de processus s'exécutant en parallèle. Toutefois, la plupart de ces algèbres partagent un ensemble de caractéristiques :

- Au niveau de la syntaxe, la plupart des algèbres sont basées sur un ensemble restreint de constructions syntaxiques permettant de définir de simples systèmes puis de les composer, grâce à quelques opérateurs, pour en générer de plus grands.
- Au niveau de la sémantique, la plupart des algèbres sont généralement équipées d'une sémantique opérationnelle structurelle de Plotkin [21]. Ce type de sémantique précise chaque pas d'exécution d'un système composé de plusieurs processus s'exécutant en parallèle.
- Finalement, pour la plupart des algèbres, on définit des relations d'équivalence permettant de capturer la notion de processus ayant le même comportement ; on définit aussi des relations d'ordre afin de pouvoir ordonner les différents processus.

Cette section couvre trois algèbres de processus permettant de modéliser des systèmes basés sur des processus. La première algèbre, CCS, est sans contredit l'algèbre la plus connue et la plus citée dans la littérature. Elle permet de se focaliser sur la définition de processus et leurs synchronisations. La deuxième algèbre est le π-calcul ; elle ajoute au calcul CCS des primitives permettant de véhiculer des valeurs sur des canaux de communication. Finalement, le calcul ambiant ajoute aux deux précédentes algèbres la possibilité de déplacer (migrer) des ressources, d'un contenant à un autre.

1.1.1 CCS

CCS [14] est une algèbre de processus permettant de modéliser un ensemble de processus, effectuant des actions plus ou moins complexes. L'utilité des cette algèbre est de pouvoir décrire un système parallèle avec un niveau d'abstraction assez élevé. On ne se focalise que sur la structure et le comportement général des processus s'exécutant en parallèle. L'utilisation d'un langage formel pour la description des systèmes offre aussi la possibilité de valider le respect d'un système par rapport à certaines propriétés souhaitées.

Syntaxe

CCS décrit des processus de façon algébrique en considérant des actions simples (non décomposables) comme éléments primitifs ; on parle alors d'actions atomiques.

Definition 1 Soit un ensemble $\mathcal{A} = \{a, b, c, ...\}$ d'*actions* atomiques. Soit $\overline{\mathcal{A}} = \{\overline{a}, \overline{b}, \overline{c}, ...\}$ l'ensemble des *co-actions*. Soit τ, appelée *action silencieuse*, l'action représentant la synchronisation entre deux processus. L'alphabet des actions correspond à $Act = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$.

Les actions α et $\overline{\alpha}$ sont dites *complémentaires*. Par exemple, en considérant un distributeur automatique de sodas, l'action *un_ dollar* pourrait représenter l'action d'un utilisateur de la machine tentant d'y introduire une pièce d'un dollar. De manière duale, la co-action $\overline{un\_\ dollar}$ représenterait alors l'acceptation par la machine d'une pièce d'un dollar. Ainsi, les deux processus en jeu, l'utilisateur et la machine pourront effectuer deux actions complémentaires et ainsi se synchroniser. Cette synchronisation est abstraite par l'action silencieuse τ qui représente le changement invisible du système suite aux deux actions complémentaires, *un_ dollar* et $\overline{un\_\ dollar}$.

Definition 2 Soit $\mathcal{P}$, l'ensemble des identifiants de processus. Soit P l'identifiant d'un processus. Tout $P \in \mathcal{P}$ est exprimé selon les règles de syntaxe définies au tableau 1.1.

$$P ::= \mathsf{0} \mid \alpha.P \mid P \ + \ P \mid P \mid P \mid P[f] \mid P \setminus L$$

TABLE 1.1 – Syntaxe de CCS.

Le processus $\mathsf{0}$ représente un processus inactif, soit le processus *nil*. Le processus $\alpha.P$ représente un processus qui effectue l'action α puis se comporte comme le (sous-)processus P. Deux opérateurs de composition permettent de définir des processus à partir d'autres processus :

1. L'opérateur $+$ est l'opérateur de choix nondéterministe ; le processus $P \ + \ Q$ représente un processus pouvant se comporter comme P ou Q.
2. L'opérateur $\mid$ est l'opérateur de composition parallèle ; le processus $P \mid Q$ représente le processus dans lequel les processus P et Q peuvent évoluer séparément ou se synchroniser.

$P[f]$ est le processus dans lequel une substitution f de la forme $[x \mapsto y]$ est effectuée ; il permet donc de définir, par renommage de certaines actions, des processus à partir d'autres processus. $P \setminus L$ est le processus ayant une restriction sur l'ensemble L d'actions. Il permet de cacher (d'abstraire) certaines actions d'un processus donné.

À cette syntaxe des processus, il faut ajouter la possibilité de définir des processus en leur attribuant un identifiant :

$$A := P$$

Voici un exemple de définition d'un processus H_0 effectuant les deux actions *tic* et *tac* :

$$H_0 := tic.tac.\mathsf{0}$$

Ce processus effectue donc l'action *tic*, puis l'action *tac* avant de s'arrêter.

Pour exprimer qu'un processus P effectue une action α pour devenir un autre processus P', on utilise la notation suivante :

$$P \xrightarrow{\alpha} P'$$

Ainsi, le processus H_0 évolue de la manière suivante :

$$H_0 \xrightarrow{tic} tac.\mathsf{0} \xrightarrow{tac} \mathsf{0}$$

Par ailleurs, pour exprimer qu'un processus effectue des actions perpétuellement, on le définit de manière récursive :

$$H := tic.tac.H$$

Ce processus ne s'arrêtera jamais :

$$H \xrightarrow{tic} tac.H \xrightarrow{tac} H \xrightarrow{tic} \dots$$

Si deux processus s'exécutent en parallèle :

$$P := a.P_1 \mid b.P_2$$

alors, soit on a :

- $P \xrightarrow{a} P_1 \mid b.P_2$
  On considère dans ce cas que c'est le sous-processus $a.P_1$ qui a évolué.
- $P \xrightarrow{b} a.P_1 \mid P_2$
  On considère dans ce cas que c'est le sous-processus $b.P_2$ qui a évolué.
- $P \xrightarrow{\tau} P_1 \mid P_2$
  On considère dans le cas que b et a sont complémentaires, c'est-à-dire : $b = \overline{a}$. Les deux sous-processus évoluent en se synchronisant sur leur action complémentaire.

Pour mieux exprimer l'évolution d'un processus, il est nécessaire d'utiliser une notation formelle permettant de définir de manière plus précise la relation «$\_ \xrightarrow{=} \_$».

Sémantique opérationnelle

La sémantique opérationnelle de CCS est présentée au tableau 1.2 ; elle définit de façon formelle le rôle de chaque opérateur et constante de l'algèbre.

Voici une description informelle de ces différentes règles :

$$\frac{\square}{\alpha.P \xrightarrow{\alpha} P} \ (\mathsf{pre})$$

$$\frac{P \xrightarrow{\alpha} Q \qquad A := P}{A \xrightarrow{\alpha} Q} \ (\mathsf{def})$$

$$\frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \ (\mathsf{comp}_1)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P + Q \xrightarrow{\alpha} Q'} \ (\mathsf{comp}_2)$$

$$\frac{P \xrightarrow{\alpha} P'}{P \mid Q \xrightarrow{\alpha} P' \mid Q} \ (\mathsf{ent}_1)$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P \mid Q \xrightarrow{\alpha} P \mid Q'} \ (\mathsf{ent}_2)$$

$$\frac{P \xrightarrow{\alpha} P' \qquad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \ (\mathsf{sync})$$

$$\frac{P \xrightarrow{\alpha} P' \qquad \alpha \notin L \cup \overline{L}}{P \setminus L \xrightarrow{\alpha} P'} \ (\mathsf{rest})$$

$$\frac{P \xrightarrow{\alpha} P' \qquad \beta = f(\alpha)}{P[f] \xrightarrow{\beta} P'[f]} \ (\mathsf{ren})$$

TABLE 1.2 – Sémantique opérationnelle de CCS.

- la règle (pre) précise de manière formelle comment le processus $\alpha.P$ peut évoluer en P en effectuant l'action α ;
- la règle (def) précise que $A := P$ n'est qu'une association entre l'identifiant A et le processus P ; si ce dernier évolue par une action α en un processus Q, il en va de même pour A ;
- les règles (comp_1) et (comp_2) précisent que le processus $P + Q$ peut soit évoluer en un processus P', résultant de l'évolution du processus P, soit en un processus Q', résultant de l'évolution du processus Q ;
- les règles (ent_1) et (ent_2) précisent que le processus $P \mid Q$ permet aux processus P et Q d'évoluer de manière indépendante (en parallèle) ;
- la règle (sync) précise que des processus P et Q, s'exécutant en parallèle, peuvent se synchroniser si elles effectuent des actions complémentaires ;
- la règle (rest) précise que le processus $P \setminus L$ ne peut évoluer que par une action α qui n'est ni membre de L ni de $\overline{L}$;
- finalement, la règle (ren) précise comment évolue un processus défini à partir d'un autre processus par renommage de certaines de ses actions.

Exemple Pour illustrer l'utilisation de ces règles, considérons un exemple dans lequel deux programmes interagissent, de manière parallèle, avec un processus qui abstrait le comportement d'un sémaphore d'exclusion mutuelle :

- Voici la définition du processus *Sem* :

  $$Sem := p.v.Sem$$

  Rappelons que les actions p et v permettent respectivement d'accéder au sémaphore et de libérer cet accès.
- Les deux programmes qui vont se disputer l'accès simultané au sémaphore ont un comportement identique moyennant leurs actions privées effectuées une fois l'accès au sémaphore acquis. À cette fin, nous définissons un processus P comme suit :

  $$P := \overline{p}.a.b.\overline{v}.P$$

  Ainsi, le comportement typique d'un tel processus est de se synchroniser avec le sémaphore grâce à l'action $\overline{p}$, complémentaire de l'action p, d'effectuer ses deux actions, a et b, puis de se synchroniser de nouveau avec le sémaphore pour en libérer l'accès.
  À partir du processus P, il est assez simple de définir deux processus P_1 et P_2 par simple renommage :

  $$P_1 := P[a \mapsto a_1,\, b \mapsto b_1]$$
  $$P_2 := P[a \mapsto a_2,\, b \mapsto b_2]$$

Ces définitions, par renommage, sont équivalentes aux définitions suivantes :

$$P_1 := \overline{p}.a_1.b_1.\overline{v}.P_1$$
$$P_2 := \overline{p}.a_2.b_2.\overline{v}.P_2$$

– Finalement, le système au complet est défini comme suit :

$$S := (P_1 \mid Sem \mid P_2) \setminus \{p, v\}$$

Ainsi, le système S correspond effectivement à deux processus P_1 et P_2 s'exécutant en parallèle avec un processus *Sem*, représentant un sémaphore, et se disputant l'accès à ce sémaphore. La restriction « $\setminus\{p, v\}$ » permet de contraindre le processus S à exécuter n'importe quelle action autre que p, v, $\overline{p}$, et $\overline{v}$. En effet, ces actions ne pourront être exécutées qu'en interne lors des synchronisations.

À partir de ces définitions et des règles de la sémantique opérationnelle, il est assez simple de pouvoir prédire certaines évolutions possibles du système (processus) S :

$$\begin{aligned}
& S \\
= \quad & \langle \text{ Par définition de } S \rangle \\
& (P_1 \mid Sem \mid P_2) \setminus \{p, v\} \\
= \quad & \langle \text{ Par définition de } P_1 \text{ et } Sem \rangle \\
& (\overline{p}.a_1.b_1.\overline{v}.P_1 \mid p.v.Sem \mid P_2) \setminus \{p, v\} \\
\xrightarrow{\tau} \quad & \langle \ P_1 \text{ et } Sem \text{ se synchronisent } \rangle \\
& (a_1.b_1.\overline{v}.P_1 \mid v.Sem \mid P_2) \setminus \{p, v\} \\
\xrightarrow{a_1} \quad & \langle \ P_1 \text{ effectue son action privée } a_1 \ \rangle \\
& (b_1.\overline{v}.P_1 \mid v.Sem \mid P_2) \setminus \{p, v\} \\
\xrightarrow{b_1} \quad & \langle \ P_1 \text{ effectue son action privée } b_1 \ \rangle \\
& (\overline{v}.P_1 \mid v.Sem \mid P_2) \setminus \{p, v\} \\
\xrightarrow{\tau} \quad & \langle \ P_1 \text{ et } Sem \text{ se synchronisent } \rangle \\
& (P_1 \mid Sem \mid P_2) \setminus \{p, v\}
\end{aligned}$$

Ainsi, le système S peut effectuer la trace $\tau.a_1.b_1.\tau$ et revenir à son état initial. Cette trace abstrait le fait que P_1 a pu se synchroniser avec le sémaphore et qu'il a effectué ces deux actions privées a_1 et a_2. Les deux occurrences de l'action silencieuse τ abstraient les deux étapes de synchronisation avec le sémaphore. Comme on peut le constater, à aucun moment, le processus S effectue l'action p et v ou leur version complémentaire.

Pour terminer, notons qu'il est d'usage de construire, à partir des règles, un arbre de dérivation qui prouve qu'un processus peut évoluer vers un autre processus.

Résumé

En résumé, CCS est une algèbre assez simple. Elle permet de modéliser les relations entre processus mis en parallèle. Une relation est représentée par la synchronisation d'une action effectuée par un processus et d'une co-action (action inverse) effectuée par un deuxième processus. Cette synchronisation permet alors aux deux processus d'évoluer. Ce modèle représente une forme assez limitée de communication ; aucune valeur ne peut être échangée entre deux processus.

1.1.2 π-calcul

Le π-calcul [16] est une algèbre de processus basée sur CCS, mais qui améliore la notion de communication entre processus. Il est possible d'échanger des valeurs, mais aussi des canaux de communication, ce qui permet de changer les relations entre les processus.

Syntaxe

La syntaxe du π-calcul ressemble fortement à celle de CCS. La différence principale se trouve dans la communication entre les processus placés en parallèle. Une notion de variable est ajoutée à la communication, ce qui permet d'échanger les canaux de communication ou toute autre donnée. Les règles de la syntaxe du π-calcul sont définies dans le tableau 1.3.

$$P ::= 0 \mid P_1 + P_2 \mid \overline{y}x.P \mid \mathrm{y(x).P} \mid \tau.P \mid P_1 \mid P_2 \mid \mathrm{(x)P}$$

TABLE 1.3 – Syntaxe du π-calcul.

Tout comme pour CCS, 0 représente un processus inactif. La composition de processus est aussi définie à l'aide des opérateurs $+$, qui représente le choix nondéterministe, et $|$, qui représente la mise en parallèle de processus. Soit les préfixes τ, $\overline{y}x.$ et $y(x).$:

– τ représente l'action silencieuse. $\tau.P$ effectue l'action τ et se comporte ensuite comme P.

- $\overline{y}x.$ est le préfixe négatif. Il représente le port de sortie du canal de communication y. Pour le processus $\overline{y}x.P$, la valeur x est envoyée sur le canal y et le processus se comporte ensuite comme P.
- $y(x).$ est le préfixe positif. Il représente le port d'entrée du canal de communication y. Pour le processus $y(x).P$, la valeur x est lue sur le canal y et le processus se comporte ensuite comme $P\{z/x\}$ (P pour lequel la variable x a été substituée par la valeur z).

La restriction $(x)P$ interdit la communication sur le canal x entre le processus restreint et un autre processus mis en parallèle.

Exemple

Voici un exemple de l'échange d'un canal de communication par un autre canal de communication. Soit les processus suivants :

$$P := \overline{b}a.S$$
$$Q := b(c).\overline{c}5.T$$
$$R := a(e).U$$

Un canal de communication b est partagé entre P et Q. P peut tout d'abord utiliser ce canal pour envoyer un canal a au processus Q. Une fois que Q reçois ce canal, il l'utilise pour envoyer la valeur 5 au processus R. Voici les étapes de ces échanges :

1. La mise en parallèle de ces processus donne ceci :

$$\overline{b}a.S \mid b(c).\overline{c}5.T \mid a(e).U$$

2. La première action possible est l'échange du canal a via le canal b :

$$\overline{b}a.S \mid b(c).\overline{c}5.T \mid a(e).U \xrightarrow{\tau} S \mid \overline{a}5.T \mid a(e).U$$

   Lors de cet échange, la substitution $(\overline{c}5.T)\{a/c\}$ survient dans le processus Q pour qu'il devienne $\overline{a}5.T$. Il peut maintenant utiliser le canal a.

3. La dernière étape l'échange de la valeur 5 entre le processus Q et le processus R, via le canal a :

$$S \mid \overline{a}5.T \mid a(e).U \xrightarrow{\tau} S \mid T \mid U$$

   Suite a cet échange, la substitution $(U)\{5/e\}$ survient.

Résumé

En résumé, les ajouts du π-calcul par rapport à CCS sont assez restreints. L'ajout de canaux de communication permet maintenant d'échanger des valeurs ainsi que des canaux de communication entre processus. Les relations entre ceux-ci peuvent donc être modifiées. Il existe quelques extensions du π-calcul permettant d'ajouter certaines possibilités. Par exemple, le π-calcul polyadique [15] permet d'échanger des listes de valeurs entre processus lors d'une communication. Une autre version du π-calcul, nommée π-calcul d'ordre supérieur [33] permet le passage de processus entier par un canal de communication, ce qui leur permet une certaine mobilité.

1.1.3 Calcul ambiant

Le calcul ambiant [5] est un calcul qui a été créé en lien avec le fort besoin de mobilité induit par l'avènement d'Internet. Le concept de base de ce calcul est l'*ambiant*. Il représente un contenant dans lequel des processus sont exécutés et dans lequel d'autres ambiants peuvent se trouver. Les ambiants peuvent donc être utilisés pour représenter à la fois des processus, des ensembles de processus, des ordinateurs sur un réseau, etc.

Syntaxe

La syntaxe du calcul ambiant ressemble beaucoup à celle de CCS et celle du π-calcul, mais comporte tout de même plusieurs nouveautés. Le concept d'ambiant sur lesquels des opérations (capacités) sont possibles est introduit. La présentation de cette syntaxe est découpée en deux parties :

1. La première partie décrit la syntaxe des processus. Elle contient les actions qu'un processus peut faire.
2. La deuxième partie décrit la syntaxe des capacités. Les capacités sont les actions qu'il est possible d'effectuer sur les ambiants.

$$P ::= 0 \mid n[P] \mid P \mid Q \mid M.P \mid (n).P \mid \langle M \rangle \mid !P \mid (\nu n)P$$

TABLE 1.4 – Syntaxe des processus du calcul ambiant.

La syntaxe des processus est présentée dans le tableau 1.4. Dans un processus, 0 représente le processus inactif. Un ambiant nommé n qui contient un processus P est

représenté par $n[P]$. La composition de processus est définie par $P \mid Q$. $M.P$ est l'exécution de la capacité M sur le processus P. L'envoi d'une capacité M de façon asynchrone est représenté par le processus $\langle M \rangle$. De son côté, la réception de façon synchrone d'une capacité n suivie de l'exécution de P est représentée par $(n).P$. La duplication de processus est faite à l'aide de $!P$. Le résultat d'une duplication de processus est $P \mid !P$. Finalement, la restriction est représentée par $(\nu n)P$. Contrairement au π-calcul, la restriction n'est pas sur le nom d'un canal, mais sur le nom d'un ambiant.

$$M ::= x \mid n \mid \text{in } M \mid \text{out } M \mid \text{open } M \mid \epsilon \mid M.M'$$

TABLE 1.5 – Syntaxe des capacités du calcul ambiant.

La syntaxe des capacités est présentée dans le tableau 1.5. Une variable est représentée par x. Une valeur lui est attribuée lors d'une réception synchrone $(x).P$. La capacité n est utilisée pour créer un ambiant nommé n. La capacité d'entrée $in\ M.P$ est utilisé pour faire entrer le processus P à l'intérieur l'ambiant M. La capacité de sortie $out\ M.P$ est utilisée pour faire sortir le processus P de l'ambiant M. La capacité d'ouverture $open\ M.P$ est utilisée pour dissoudre l'ambiant M et se comporter ensuite comme le processus P. La capacité vide (qui ne fait rien) est représentée par ϵ. Finalement, $M.M'$ est la séquence de capacités.

Exemple

Contrairement au π-calcul, les canaux de communications nommés n'existent pas dans le calcul ambiant. Lorsqu'une telle structure est nécessaire, un ambiant qui sert à la communication doit être utilisé pour échanger la valeur entre deux ambiants servant à effectuer les traitements. Voici un exemple de l'échange d'une valeur entre deux ambiants :

$$Env[Msg[out\ Env.in\ Recv.0 \mid \langle v \rangle] \mid Q] \mid (\nu Msg)Recv[open\ Msg.0 \mid (x).R]$$

La réduction de cet exemple se fait par les étapes suivantes :

1. L'ambiant message Msg sort tout d'abord de l'ambiant envoyeur Env :

$$Msg[in\ Recv.0 \mid \langle v \rangle] \mid Env[Q] \mid (\nu Msg)Recv[open\ Msg.0 \mid (x).R]$$

2. L'ambiant message entre ensuite à l'intérieur de l'ambiant receveur :

$$Env[Q] \mid (\nu Msg)Recv[Msg[0 \mid \langle v \rangle] \mid open\ Msg.0 \mid (x).R]$$

3. À l'intérieur de l'ambiant receveur $Recv$, la capacité *open* Msg peut maintenant être exécutée pour ouvrir le message. Le message ne pouvait être ouvert avant d'entrer dans l'ambiant $Recv$ à cause de la restriction (νMsg) sur celui-ci :

$$Env[Q] \mid (\nu Msg)Recv[\langle v \rangle \mid 0 \mid (x).R]$$

4. Le message peut maintenant être envoyé au processus R, dans lequel la variable x est substituée par la valeur v :

$$Env[Q] \mid (\nu Msg)Recv[R\{v/x\}]$$

Résumé

En résumé, le calcul ambiant ajoute plusieurs possibilités par rapport à CCS et au π-calcul. Le concept d'ambiant est tout d'abord introduit pour représenter différents types de contenants dans lesquels des processus peuvent être exécutés. Contrairement au π-calcul, les canaux de communication nommés n'existent pas. Il est tout de même possible de reproduire leur comportement à l'aide d'un ambiant qui sort d'un ambiant envoyeur pour entrer dans un ambiant receveur. Finalement, les opérations sur les ambiants introduisent la mobilité de processus. Ce concept, qui n'existe pas dans les deux autres algèbres présentées précédemment, donne un plus grand contrôle sur la structure des processus.

1.2 Langages concurrents

Dans cette section, des langages de programmation concurrents, ainsi que quelques extensions de langage permettant d'ajouter des notions de concurrence à un langage sont présentés. Ces langages et extensions se ressemblent par leur support de la majorité des concepts présentés précédemment avec les algèbres de processus. Ils sont par contre plus complexes que ces algèbres, car ils doivent supporter une réalité qui est moins abstraite et qui doit tenir compte de plusieurs aspects ; par exemple, la mémoire partagée, les verrous, les entrées/sorties, etc. Suite à la description de chacun de ces langages et extensions, un exemple basé sur l'exemple du serveur *pong* tiré de [8] est présenté pour monter l'interaction entre deux processus. Dans les exemples, un processus représentant un serveur peut recevoir un message *ping* et répond par un message *pong* à l'envoyeur.

Bien que chacun de ces langages et extensions de langage ait ses particularités, ils

permettent tous de créer des processus concurrents et un moyen de communication existe entre ces processus. Les langages suivants sont présentés :

- Erlang est un langage fonctionnel permettant de créer beaucoup de processus légers. Ces processus peuvent être démarrés de façon locale ou à distance. Une relation parent/enfant existe entre un processus et un autre processus qu'il a créé ; elle est utilisée pour la gestion des exceptions. La communication se fait à l'aide d'une boîte aux lettres sur chacun des processus.
- Termite Scheme est une extension du langage fonctionnel Gambit Scheme ajoutant des notions de distribution de processus à ce dernier. Tout comme Erlang, les processus peuvent être démarrés à distance, mais une migration de processus en cours d'exécution est aussi possible entre deux interpréteurs. La communication se fait aussi à l'aide d'une boîte aux lettres.
- JCSP.net est une implémentation de l'algèbre de processus CSP sous la forme d'une bibliothèque pour le langage objet Java. Cette bibliothèque permet l'exécution de différents processus en parallèle et rend disponibles des canaux de communications qui peuvent aussi être utilisés entre deux machines virtuelles. Tout objet sérialisable peut être envoyé par ces canaux.
- JoCaml est un langage fonctionnel qui implémente l'algèbre de processus Join-Calculus. Il permet de démarrer en parallèle plusieurs processus et la communication se fait à l'aide de canaux de communications pouvant être synchrones ou asynchrones. La particularité de ces canaux est qu'ils sont eux-même des processus. Un canal asynchrone effectue un traitement sans retourner de valeur et un canal synchrone retourne une valeur. Ce langage supporte l'accès à distance à des canaux ainsi que la migration de processus en cours d'exécution.
- Stackless Python est une version alternative de l'interpréteur du langage objet Python. Il ajoute une notion de processus légers nommés *tasklet*, exécutés en concurrence. L'état d'exécution d'un tasklet peut être sauvegardé pour être rechargé plus tard, pas nécessairement sur la même instance de l'interpréteur. Des canaux de communications bidirectionnels sont utilisés pour les échanges entre les tasklets.

1.2.1 Erlang

Erlang [3] est un langage concurrent et fonctionnel à usage général. Il a été développé en 1986 par Joe Armstrong, chez Ericsson, pour créer des applications distribuées, en temps réel, avec une forte tolérance aux pannes et une forte disponibilité. Ce langage supporte aussi la mise à jour *à chaud* du code d'un programme sans avoir à interrompre son exécution. Erlang a d'abord été utilisé pour programmer des commutateurs téléphoniques, dans le but d'avoir une fiabilité extrême et d'éviter les pannes. La littérature mentionne souvent le commutateur téléphonique *AXD301* d'Ericsson [1] [2]

comme étant un des plus gros projets ayant utilisé Erlang.

Description d'Erlang

Le langage Erlang est basé sur un modèle par acteurs [10]. Le modèle par acteurs est un modèle mathématique pour décrire des traitements concurrents. Un acteur est vu comme une primitive universelle pour décrire un élément de traitement. La communication entre les acteurs se fait par l'échange de messages. En réponse à un message, un acteur peut effectuer un traitement, créer un autre acteur, envoyer un autre message ou déterminer comment répondre au prochain message qui sera reçu.

Dans le modèle par acteurs, les communications se font de façon asynchrone. La communication asynchrone implique qu'un processus qui envoie un message n'attend pas sa réception par un autre processus pour continuer son traitement. Chaque acteur (processus) est identifié par une adresse servant à l'échange de messages. Un processus dispose d'une boîte aux lettres pour la réception des messages. L'ordre d'arrivée des messages dans la boîte aux lettres n'est pas important. Pour lire un message, le processus récepteur utilise une recherche par patron (*pattern matching* en anglais). Le premier message filtré par le patron est lu et est retiré de la boîte. Un message peut être n'importe quelle structure Erlang : primitive (entiers, flottants, caractères, atomes), n-uplet (*tuple* en anglais), liste, fonction et même l'identifiant d'un processus. Le fait de pouvoir échanger des identifiants de processus permet de modifier la topologie du réseau de processus interconnectés.

Les processus en Erlang correspondent à des fils d'exécution légers, gérés par une machine virtuelle. Ils sont appelés ainsi par opposition aux fils d'exécution « lourds », gérés par le système d'exploitation. En effet, lorsqu'un grand nombre de fils d'exécution gérés par le système d'exploitation sont créés, leur performance d'exécution se dégrade très rapidement à cause des changements de contexte du processeur pour exécuter chacun d'eux. Les fils d'exécution légers sont habituellement exécutés sur un seul processus du système d'exploitation, ce qui permet d'en créer beaucoup plus sans affecter les performances du système. Depuis peu, Erlang supporte plusieurs fils d'exécution du système d'exploitation pour exécuter ses fils légers. Ceci permet d'exploiter plusieurs processeurs.

Les processus légers sont exécutés sur une machine virtuelle appelée nœud. Plusieurs nœuds, exécutés sur des systèmes différents, peuvent être connectés ensemble. Le démarrage d'un processus parallèle est fait à l'aide d'une commande *spawn*. Lorsque cette commande est appelée par un processus, un deuxième processus léger est lancé

en parallèle à partir d'une fonction donnée en paramètre à la commande *spawn*. Le processus qui en démarre un autre est vu comme son parent et il peut communiquer avec le processus enfant à l'aide son identifiant, retourné par *spawn*. Un processus peut être démarré sur le même nœud ou sur un nœud distant. Dans les deux cas, l'échange de message se fait de la même façon.

La gestion des erreurs liées aux processus est simple. Si un processus plante, il termine correctement son exécution, en prenant soin d'envoyer un message au processus parent qui peut prendre une action ; par exemple, redémarrer le processus interrompu.

Exemple

Cet exemple démontre le démarrage à distance de processus. Pour ce, une variante de la fonction *spawn* acceptant quatre paramètres est utilisée. Le premier paramètre sert à spécifier le nom d'une instance de l'interpréteur Erlang et le nom d'hôte du système sur lequel cette instance existe, pour démarrer le processus sur cette instance :

```
-module(pingpong).
-export([start/0, pong_server/0]).

start() ->
    Pid = spawn('b@host2',pingpong,pong_server,[]),
    Pid ! {self(), 'ping'},
    receive
      Msg ->
        io:format('~w~n',[Msg])
    end.

pong_server() ->
    receive
        {Pid,'ping'} ->
            io:format(user,'ping~n',[])
            Pid ! 'pong'
            pong_server()
    end.
```

Lors de l'appel à la méthode *start*, un processus enfant est démarré sur l'instance de l'interpréteur identifiée par *b@host2*, à partir de la fonction *pong_ server*. L'identifiant de ce processus est retourné par *spawn*. Cet identifiant est ensuite utilisé pour envoyer un

message au processus enfant, contenant l'identifiant du processus parent et le message *ping*. Suite à l'envoi, le processus parent attend une réponse du processus enfant exécuté à distance.

De son côté, le processus enfant attend un message correspondant au patron *{Pid,'ping'}*. Si le message reçu correspond, il envoie le message *pong* à l'identifiant de processus reçu (celui du processus parent). La sortie suivante est affichée sur le système sur lequel le programme est démarré :

```
host1>erl -sname a
a@host1>c(pingpong).
a@host1>pingpong:start().
a@host1>
a@host1>pong
```

La sortie suivante est affichée sur le système sur lequel le processus enfant est démarré à distance (fonction *pong_server*) :

```
host2>erl -sname b
b@host2>c(pingpong).
b@host2>
b@host2>ping
b@host2>
```

Résumé

En résumé, Erlang est un langage conçu principalement pour le parallélisme et la distribution. Les traitements sont démarrés à l'intérieur de plusieurs processus et aucune donnée n'est partagée. L'échange de données se fait à l'aide de l'envoi de messages et la réception des messages est effectuée à l'aide de recherche de patron. Un processus peut donc décider des messages qu'il veut recevoir et ignorer ceux qui ne correspondent pas aux patrons qu'il supporte. Les liens sont aussi possibles entre plusieurs nœuds, exécutés sur différents ordinateurs. Le langage permet le démarrage à distance de processus, sur un autre nœud. La communication avec les processus à distance se fait de la même façon qu'une communication locale, à l'aide de l'identifiant du processus. Un identifiant de processus peut aussi être envoyé à l'intérieur d'un message, ce qui permet de créer de nouveaux liens.

1.2.2 Termite Scheme

Termite Scheme [8] est une extension expérimentale du langage Gambit Scheme dédiée au calcul distribué. Cette extension est aussi basée sur le modèle par acteurs, mais ajoute deux concepts intéressants par rapport à Erlang. Ces deux concepts sont les macros et les continuations, qui peuvent être envoyées en tant que message. Ceci permet d'abstraire les données et les processus et offre donc des possibilités par rapport à la migration de processus en cours d'exécution et la mise à jour de code de façon dynamique.

Description de Termite Scheme

Le modèle de concurrence de Termite Scheme est basé sur un ensemble de processus identifiables, de façon unique, sur l'ensemble d'un système distribué. Ces processus sont légers et on devrait pouvoir en créer des centaines sans avoir à se soucier de la performance. Ils sont vus comme un concept important pour la modélisation des systèmes distribués. Les processus sont exécutés sur des nœuds qui sont des machines virtuelles et, à l'instar d'Erlang, un processus enfant est démarré par un processus parent à l'aide de la commande *spawn*. Les processus ont un identifiant unique à l'intérieur d'un nœud, mais aussi unique à l'intérieur d'un réseau de nœuds. Des processus peuvent être démarrés à distance sur un autre nœud. Ce démarrage à distance se fait à l'aide de la commande *remote-spawn*.

Pour ce qui est de l'échange d'information entre les processus, une isolation forte a été mise en place entre chacun d'eux. Un processus ne peut pas accéder à l'espace mémoire d'un autre processus, ce qui rend impossible la création de zones mémoires partagées. Ceci élimine donc les problèmes reliés au partage de la mémoire et à la gestion des accès exclusifs à certaines zones. Pour s'échanger des informations, les processus doivent donc utiliser des messages. À l'instar d'Erlang chaque processus a une boîte aux lettres qui lui est propre. Les messages sont stockés dans cette boîte dans l'ordre où ils sont reçus. Un message peut être de n'importe quel type atomique sérialisable ou de n'importe quel type composé, tant qu'il ne contient que des valeurs sérialisables. L'envoi de messages se fait de façon asynchrone tandis que la réception se fait de façon synchrone. Un processus qui tente de lire dans la boîte de réception alors qu'il n'y a aucun message qui correspond à ce qu'il recherche bloque. La recherche de message se fait à l'aide de prédicat ou à l'aide de recherche de patrons. Lors d'une demande de réception de message, le premier message correspondant à ce qui est demandé est retourné.

Dans un système distribué, la gestion d'erreurs et de défaillances devient très importante. Cette gestion se fait au niveau de la réception de messages. Lorsqu'un processus tente de lire un message, il est possible que celui qui doit l'envoyer n'ait pas pu l'envoyer. On peut donc spécifier un temps d'attente maximal pour recevoir un message. Si le message attendu n'arrive pas dans le délai donné, le processus débloque et peut continuer son exécution. Une gestion des erreurs peut se faire aussi à l'aide d'exceptions lancées par le programme. Lorsqu'un processus enfant est lancé à l'aide de la commande *spawn-link*, les processus parent et enfant sont connectés de façon bidirectionnelle. Les exceptions lancées par le processus enfant peuvent alors se rendre jusqu'au parent qui peut prendre une action conséquente. Pour le démarrage d'un processus sur un nœud distant, on peut utiliser *remote-spawn-link* pour tirer profit du même système de passage des exceptions au parent.

Une partie intéressante de Termite Scheme est l'utilisation des continuations pour abstraire l'exécution d'un programme. Les continuations permettent à un processus de sauvegarder son état d'exécution et de le recharger plus tard. Une continuation peut être envoyée comme message à un autre processus. Ce concept devient très intéressant, car il permet alors le déplacement des processus d'un nœud à un autre. En plus de la migration des processus, les continuations permettent aussi de les cloner et même de mettre à jour leur code pendant qu'ils sont en cours d'exécution, comme Erlang le permet.

Bien que cette migration soit rendue possible, il y a tout de même des problèmes importants à considérer. Le problème principal est lié à la gestion de la boîte aux lettres lors de la migration du processus. Deux solutions sont proposées :

- On peut abandonner son ancienne boîte aux lettres et recréer les liens de communication, au besoin, une fois le processus migré.
- Un processus mandataire (*proxy* en anglais) peut être créé sur l'ancien nœud pour rediriger tous les messages vers le nouveau nœud où se trouve le processus.

Un autre problème par rapport à la migration est lié à l'accès aux entrées/sorties. Comment doit-on gérer l'accès aux ressources qu'on ne peut migrer simplement, comme c'est le cas pour le clavier, la souris, etc. ? Encore une fois, une fois le processus migré, il peut continuer d'utiliser ces entrées/sorties à l'aide de processus mandataires pour rediriger les données.

Exemples

Voici trois exemples sur Termite Scheme tirés de [8]. Ces exemples permettent de comprendre la recherche de patron, le démarrage de processus à distance et la migration

de processus.

Le premier exemple porte sur la recherche de patron lors de la réception d'un message. Un processus serveur est démarré pour attendre la réception de messages *ping*. Lorsqu'il reçoit un tel message, il répond par le message *pong* :

```
(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from 'ping) ; pattern to match
           (where (pid? from)) ; constraint
           (! from 'pong))) ; action
        (loop))))))
```

Une fois le processus serveur démarré, son identifiant est assigné à la variable *pong-server*. Cette variable peut par la suite être utilisée pour envoyer un message au serveur. Suite à son démarrage, il effectue des appels à *recv* pour lire les messages de sa boîte aux lettres. Cet appel bloque tant et aussi longtemps que le patron *(from 'ping)* n'est pas détecté. Ce patron correspond à une liste de deux éléments, contenant un premier élément arbitraire et un deuxième élément devant être exactement le message *'ping*. Une condition est ajoutée à ce patron. Le premier élément, *from*, doit être de type *identifiant de processus* (*pid ?*). Une fois le message validé avec le patron, une réponse *'pong* est envoyée au processus ayant envoyé son identifiant à même le message original. L'envoi d'un message se fait à l'aide de l'opérateur « ! », qui reçoit en argument l'identifiant du processus auquel envoyer le message, ainsi que le message à envoyer.

Du côté du client, une requête contenant une liste de deux éléments *(list (self) 'ping)* est envoyée au serveur à l'aide de son identifiant *pong-server*. Le premier élément de la liste, *(self)*, est l'identifiant du processus client que le serveur utilise pour envoyer la réponse et le deuxième élément est le message *'ping* :

```
(! pong-server (list (self) 'ping))
```

Le deuxième exemple porte sur le démarrage à distance de processus. Un processus est démarré sur le nœud identifié par le nom d'hôte *example.com* et le port *3000*. Une fois démarré, ce processus utilise l'identifiant du processus à partir duquel il a été démarré pour lui envoyer le message *'boo* :

```
(define node (make-node "example.com" 3000))
```

```
(let ((me (self)))
  (remote-spawn node
    (lambda ()
      (! me 'boo))))

(?)
```

Dans cet exemple, un identifiant de nœud est créé avec un appel à *make-node*. Ce nœud doit déjà être actif pour recevoir un processus. L'identifiant du nœud d'origine, *(self)*, est noté dans la variable *me*. Un processus distant est ensuite démarré sur le nœud *node* à l'aide de l'appel à *remote-spawn node*. Pour le démarrage de ce processus, une fonction *lambda* est envoyée. Elle contient le code qui est exécuté sur le nœud distant. Dans l'exemple, ce code envoie le message *'boo* au nœud identifié par la variable *me* (le nœud d'origine). Du côté du nœud d'origine, l'opérateur « ? » est utilisé pour recevoir ce message. On peut noter l'utilisation directe du contenu de la variable *me* par le processus démarré à distance, pour envoyer le message. Avec Termite Scheme, les identifiants de processus sont uniques et connus par l'ensemble des nœuds.

Le dernier exemple porte sur la migration de processus. Une nouvelle fonction nommée *migrate-task* est définie et permet de migrer le processus qui effectue un appel à celle-ci vers le noeud *node* :

```
(define (migrate-task node)
  (call/cc
    (lambda (k)
      (remote-spawn node (lambda () (k #t)))
      (halt!))))
```

Dans le but de migrer le processus, un appel à l'opérateur *call/cc* est effectué pour créer une continuation. On peut voir la continuation comme une sauvegarde de l'état d'exécution au moment où *call/cc* est appelé. Cette continuation est passée en paramètre à la fonction *lambda* qui suit. Dans cette fonction, un appel à *remote-spawn* est fait pour lancer à distance une nouvelle fonction *lambda* qui appellera la continuation pour récupérer l'état d'exécution sauvegardé. Un appel à *halt* est finalement effectué pour arrêter le processus original, ayant demandé la migration. Du côté du nœud distant, lorsque la continuation est appelée, l'exécution se poursuit après le point où *call/cc* a été appelé. Le processus continu donc son exécution au point où il était rendu.

Résumé

En résumé, les fonctionnalités de Termite Scheme ressemblent beaucoup à celle d'Erlang. La différence la plus notable est le support de continuations pouvant être envoyées à l'intérieur d'un message comme tout autre objet sérialisable. Ces continuations permettent donc la migration de processus en cours d'exécution. Une autre différence par rapport à Erlang est la relation optionnelle entre un processus parent et un processus enfant. Lorsque cette relation est nécessaire, la méthode *spawn-link* est utilisée à la place de *spawn* pour démarrer le processus enfant.

1.2.3 JCSP.net

JCSP.net [48] est une implémentation de l'algèbre de processus CSP sous forme d'une bibliothèque pour le langage Java. De plus, comme son nom l'indique (*.net*), cette bibliothèque permet de gérer des communications via un réseau pour étendre la portée de l'interaction entre les processus.

La bibliothèque JCSP.net sert principalement à faciliter la tâche de création d'un réseau de processus communicant et se synchronisant entre eux, avec le langage Java. Étant basée sur l'algèbre de processus CSP, il devient facile de vérifier si un programme respecte une formule CSP ou même de traduire des définitions de structure de programme en CSP, pour ensuite traduire cette structure en code Java utilisant cette bibliothèque.

Description de JCSP.net

En JCSP.net, un processus est un composant qui encapsule traitements et données. Ces traitements et données sont privés, c'est-à-dire qu'un processus ne peut pas voir ce qui appartient à un autre processus. Chaque processus est géré par un fil d'exécution Java, qui est traité comme un fil d'exécution du système d'exploitation. On est donc certain que plusieurs processus JCSP.net tireront profit de systèmes disposant d'un processeur multicœur ou de plusieurs processeurs.

L'échange de données entre processus JCSP.net se fait à l'aide de canaux de communication synchrones, c'est-à-dire que celui qui envoie un message reste bloqué tant qu'un processus receveur ne l'a pas demandé. Dans l'autre sens, un processus qui attend un message d'un autre processus reste bloqué tant que ce dernier ne l'a pas envoyé. Sur

une même machine virtuelle, les canaux sont tout d'abord instanciés et sont ensuite partagés entre les processus qui en ont besoin. Si plusieurs processus ont besoin de communiquer entre des machines virtuelles séparées par un réseau, un serveur de nom est nécessaire pour créer les canaux et ensuite y accéder. De plus, entre deux machines virtuelles, il n'y a qu'une seule connexion réseau qui est effectuée pour l'ensemble des canaux qui seront utilisés. Un système de multiplexage des messages a été mis en place pour qu'un message soit dirigé vers le bon canal. En plus des canaux de communication nommés, des canaux de communication anonymes peuvent être créés. Une fois un canal anonyme créé, un processus doit informer un autre processus avec qui il communique de l'existence de ce canal pour qu'ils puissent l'utiliser.

Tout comme Erlang et Termite Scheme, JCSP.net permet le démarrage de processus sur une machine virtuelle distante. Il est donc possible d'exploiter ce concept pour créer des agents mobiles qui se déplacent d'une machine à une autre. Il n'est par contre pas possible de déplacer un processus en cours d'exécution d'une machine virtuelle à une autre à cause de restrictions du langage Java. Seules les données liées aux objets Java peuvent être sérialisées. Lorsqu'un processus est démarré à distance, une gestion automatique des dépendances est faite. Toutes les classes Java que le processus à distance a besoin pour démarrer sont téléchargées à partir de la machine virtuelle qui initie le démarrage. Ceci simplifie beaucoup la tâche de migration d'un objet JCSP.net relié à un processus.

Suite à la migration de la structure d'un processus JCSP.net, il est possible de faire une reconnexion des canaux de communication précédemment utilisés. Certaines règles sont à suivre, car cette migration est vue comme dangereuse. Premièrement, la topologie du réseau de processus est changée ce qui peut amener des problèmes difficiles à détecter. De plus, lorsqu'un canal est en train d'être reconnecté, il y a un laps de temps où il faut gérer les possibles demandes de communication sur celui-ci. Le côté en écriture est simple à migrer, car la communication est directement coupée et il n'y a pas de risque de perdre des messages. Du côté de la lecture, un protocole spécial a été mis en place :

1. La première étape est de demander au serveur de noms de bloquer le canal. Ce blocage fait en sorte que toute demande d'écriture sur le canal se termine par une exception, lancée sur la machine virtuelle où se trouve le processus qui veut écrire. Cette exception est gérée directement par la bibliothèque JCSP.net et le programmeur n'a pas à s'en soucier. Le message à envoyer est mis en attente.
2. Une fois le canal bloqué, la connexion peut être migrée vers un autre processus. Le serveur de noms est alors informé du nouveau processus qui doit recevoir les messages envoyés sur le canal migré.
3. Finalement, lorsque la migration est terminée, le serveur de noms débloque le canal et la communication peut suivre son cours. Dès que le canal est débloqué,

la bibliothèque peut alors envoyer les messages en attente.

Exemple

Cet exemple est basé sur deux processus lancés séparément. Un premier processus, *ClientPing*, se trouve sur un nœud A. Il envoie un message *ping* sur un canal nommé *canal_envoi* et reçoit ensuite une réponse sur un canal nommé *canal_retour*. Cette réponse est affichée à l'écran. Du côté d'un nœud B, un processus *ServeurPong* est démarré. Ce processus écoute sur le canal *canal_envoi*. Si ce processus reçoit un message *ping*, il répond par le message *pong* sur le canal *canal_retour*.

Voici le code du processus *ClientPing*, exécuté sur le nœud A :

```
class ClientPing implements CSProcess {
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

    public ClientPing(ChannelInputInt in,
        ChannelOutputInt out)
    {
        this.in = in
        this.out = out;
    }

    public void run() {
        String reponse;

        out.write("ping");
        reponse = in.read();

        System.out.println("Réponse reçue: " +
            reponse);
    }
}
```

Dans le code du processus *ClientPing*, les deux canaux sont reçus dans le constructeur de la classe. La méthode *run* est appelée lorsque le fil d'exécution est démarré pour ce processus. Le message *ping* est envoyé sur le canal *out* et une lecture est faite sur le canal *in* pour retrouver la réponse. Cette réponse est affichée.

Pour démarrer le processus *ClientPing*, les canaux doivent être d'abord créés et le processus doit être instancié. Voici comment :

```
Net2OneChannel in = new Net2OneChannel(
    "canal_retour"
);

One2NetChannel out = new One2NetChannel(
    "canal_envoi"
);

new Parallel (
    new CSProcess[] {
        new ClientPing(in, out)
    }
).run();
```

Dans ce code, deux canaux sont déclarés. Le canal *canal_envoi* est de type *One2-NetChannel*. Un tel canal est utilisé par un processus local et envoie les valeurs sur le réseau, à l'aide de l'identifiant *canal_envoi*. Le serveur de nom gère la synchronisation entre les différentes machines virtuelles qui utilisent un tel canal réseau. Un canal de type *Net2OneChannel* est aussi créé pour recevoir la valeur de retour. Ce canal est identifié par le nom *canal_retour*. Finalement, le processus *ClientPing* est démarré et les canaux lui sont envoyés.

Du côté du nœud *B*, un processus nommé *ServeurPong* est démarré. La structure est semblable à celle du processus *ClientPing* :

```
class ServeurPong implements CSProcess {
    private final ChannelInputInt in;
    private final ChannelOutputInt out;

    public ServeurPong(ChannelInputInt in,
        ChannelOutputInt out)
    {
        this.in = in
        this.out = out;
    }

    public void run() {
        while (true) {
            String message = in.read();
```

```
                if ("ping".equals(message)) {
                    out.write("pong");
                } else {
                    out.write(null);
                }
            }
        }
    }
```

Le processus *ServeurPong* est démarré :

```
Net2OneChannel in = new Net2OneChannel(
    "canal_envoi"
);

One2NetChannel out = new One2NetChannel(
    "canal_retour"
);

new Parallel (
    new CSProcess[] {
        new ServeurPong(in, out),
    }
).run ();
```

Dans le cas de *ServeurPong*, une boucle infinie est utilisée pour pouvoir répondre à tous le message *ping* qu'il reçoit. Contrairement au démarrage de *ClientPing*, les types des canaux utilisés par *ServeurPong* sont inversés. Le canal *canal_envoi* est associé à un canal de type *Net2OneChannel*, car il est utilisé pour recevoir ce qui est envoyé par le canal *One2NetChannel* du même nom, utilisé par *ClientPing*.

Résumé

En résumé, JCSP.net permet principalement d'ajouter la notion de canaux de communication au langage Java. Des constructions permettant de démarrer les processus en parallèle, utilisant ces canaux, permettent de créer des programmes respectant l'algèbre de processus CSP. Cette structure peut être avantageuse pour la vérification formelle de programmes écrits avec cette bibliothèque.

Les canaux de communication peuvent communiquer par un réseau. Un serveur de noms doit être utilisé pour définir des canaux nommés. Différents processus, démarrés sur différentes machines virtuelles, peuvent alors utiliser un même nom de canal. La bibliothèque gère automatiquement la copie des classes Java lorsque des objets sont envoyés à un nœud qui ne connaît pas leur structure. Ce transfert de classe peut permettre le démarrage à distance de processus, mais ne permet pas la migration de processus en cours d'exécution.

1.2.4 JoCaml

JoCaml [38] [39] est un langage destiné à simplifier la création de systèmes concurrents, distribués et basés sur les agents mobiles. Ce langage est basé sur le Join-Calculus [7] qui offre des primitives simples et puissantes pour la création de processus, la communication entre eux ainsi que leur migration. Deux versions de JoCaml existent dont une qui n'est malheureusement plus maintenue. Cette section discute de cette dernière, car elle supporte la mobilité de code.

Description de JoCaml

Le modèle de JoCaml provient du Join-Calculus qui utilise un concept nommé *location* pour abstraire les processus et les systèmes hôtes. Les *locations* peuvent communiquer entre elles par des canaux. Une *location* peut être contenue à l'intérieur d'une autre *location* et peut aussi être déplacée d'une location à une autre. Lorsqu'il y a déplacement, tout le contenu de la *location* impliquée, incluant les canaux de communications, est déplacé. Tous les canaux de communication qu'il pourrait y avoir entre la *location* déplacée et les autres environnantes restent intacts. L'opérateur *go [cible]* est utilisé pour déplacer le processus courant vers la *location cible*. Ce modèle ressemble fortement à celui du calcul ambiant. Dans ce calcul, un ambiant est un contenant pouvant exécuter un traitement et sur lequel des opérations existent pour le faire entrer dans un autre ambiant ou l'en faire sortir.

Avec JoCaml, les canaux de communications sont créés à l'aide de la construction *def*. Un canal ressemble de très près à une fonction ; il a un nom, reçoit un argument, exécute un traitement et peut retourner une valeur. Le nom du canal se nomme *port*. Il est utilisé par les autres processus pour appeler le canal. Un canal est synchrone lorsqu'il retourne une valeur, à l'aide de l'opérateur *reply*. Il s'utilise alors de la même façon qu'une fonction. Un canal est asynchrone lorsqu'il ne retourne aucune valeur.

Dans ce cas, un appel à celui-ci termine immédiatement et le traitement de la valeur envoyée au canal peut avoir lieu plus tard. Le suffixe « ! » doit être ajouté au nom d'un canal asynchrone lors de sa déclaration. Lorsqu'un appel est effectué à un canal, un fil d'exécution est lancé pour effectuer le traitement associé au canal.

Plusieurs canaux peuvent être associés à un traitement. Ce modèle se nomme *patron de jointure* (*Join-pattern* en anglais). Un appel doit alors être effectué à chacun des canaux pour que le traitement soit lancé. Un tel traitement peut donc dépendre de deux valeurs reçues de deux canaux différents. Des processus exécutés en parallèle devront faire les appels sur les différents canaux.

Dans un environnement distribué, un serveur de nom (*Ns*) est utilisé pour retrouver les canaux. Ce serveur de nom offre deux méthodes, soit *register*, qui permet d'enregistrer un canal distribué, et *lookup*, qui permet de retrouver un canal distribué. Lorsqu'un processus retrouve un canal distribué, pouvant se trouver sur un autre ordinateur, il peut directement l'appeler et recevoir le résultat comme s'il faisait un appel à un canal local. Lorsque le canal de communication est appelé, un fil d'exécution est démarré pour recevoir le message et exécuter les instructions associées au nom du canal.

Exemple

Voici un exemple du serveur *pong* à l'aide de JoCaml. Étant donné que les processus de ce langage s'apparentent à des appels de fonctions, cet exemple est très léger, mais comporte tout de même quelques subtilités :

```
type message = ping
def serveur_pong(ping) = reply "pong" to pong_server
;;
```

Le processus suivant effectue un appel à *pong_server* et affiche la réponse obtenue :

```
spawn {print_string serveur_pong(ping)}
;;
```

Dans cet exemple, quelques aspects importants sont à noter. Le nom *serveur_pong* est tout d'abord le nom d'un canal qui peut recevoir un message. Un type nommé *message* est défini et comporte uniquement une valeur *ping*. Cette valeur est définie comme paramètre du canal *serveur_pong*. Ce paramètre se voit donc systématiquement assigner le type *message*. Avec cette façon de faire, il est possible de faire de la détection

de patron. Dans le cas présent, le canal *serveur_pong* ne peut être appelé que si la valeur *ping* est envoyée. L'envoi de toute autre valeur causera une erreur.

Résumé

En résumé, JoCaml offre une façon intéressante de voir les canaux de communication. Au lieu qu'un canal soit utilisé pour échanger des valeurs entre deux processus arbitraires, un canal est nécessairement associé à un processus qu'il lance lorsqu'une valeur lui est envoyée. À l'aide du *patron de jointure*, on peut même associer plusieurs canaux à un même processus ; une valeur doit être envoyée à chacun de ces canaux pour que le processus soit lancé. Cette façon de faire permet de créer des processus qui se lancent en cascade, au fur et à mesure que des résultats intermédiaires sont disponibles. Les canaux peuvent aussi être appelés à distance. De cette façon, des traitements spécifiques peuvent être exécutés sur des systèmes spécifiques. Finalement, tout comme Termite Scheme, JoCaml supporte la migration de processus.

1.2.5 Stackless Python

Stackless Python [35] est une amélioration apportée à l'implémentation officielle du langage Python. Cette amélioration apporte quelques concepts de concurrence, tout en retenant une compatibilité avec les programmes Python écrits de façon standard. En plus de supporter la base du langage Python, Stackless Python offre la possibilité ce créer des microprocessus concurrents pouvant communiquer par canaux.

Description de Stackless Python

Le principal ajout de Stackless Python est le support de microprocessus appelé *tasklet*. Ces microprocessus ont été conçus pour être les plus légers possible, de sorte que des milliers d'entre eux puissent être démarrés en parallèle sans avoir à se soucier de la performance du système. Ils sont exécutés à tour de rôle par un ordonanceur, à l'intérieur d'une machine virtuelle qui correspond à un seul processus du système d'exploitation. Ceci fait en sorte qu'un seul processeur est exploité dans un environnement multicœur ou multiprocesseur.

Concernant la communication entre tasklets, des canaux bidirectionnels sont utilisés. Comme pour JCSP.net, la communication est effectuée de façon synchrone. Un tasklet

qui envoie un message bloque tant qu'un autre tasklet n'a pas lu le message. Si plusieurs tasklets envoient des messages à un unique tasklet receveur, les messages sont lus dans l'ordre où ils ont été envoyés. À travers un canal, tout type d'objet Python peut être envoyé ; seule la référence d'un objet est échangée.

À l'intérieur d'un interpréteur Stackless Python, l'ordonnancement des tasklets est gardé à sa plus simple expression. Chaque tasklet est exécuté à tour de rôle, dans l'ordre où ils ont été démarrés. Un tasklet est exécuté tant et aussi longtemps qu'il n'a pas terminé son travail. Si on veut éviter ce mode de fonctionnement, il y a deux possibilités. Un tasklet peut donner explicitement la main au prochain en attente à l'aide de la fonction *stackless.schedule* ; sinon il est possible d'indiquer un nombre maximal d'instructions à exécuter sur chaque tasklet avant de le remettre en file d'attente et d'exécuter le même nombre d'instructions sur le prochain tasklet en attente. On indique le nombre d'instructions à exécuter lors de l'appel à *stackless.run* pour démarrer l'exécution des tasklets.

Une des principales fonctionnalités de Stackless Python est la possibilité de sérialiser les *tasklets* [36]. Ceci permet donc de sauvegarder l'état d'exécution d'un tasklet et de le restaurer plus tard. Ce qui rend possible cette sauvegarde est l'utilisation des continuations [43] pour abstraire l'exécution de chacune des instructions d'un tasklet. La restauration d'un tasklet peut se faire dans une autre instance de l'interpréteur, ce qui offre une base pour la migration de tasklets.

Exemple

Voici maintenant un exemple de processus *serveur_pong* qui répond à un message *ping*. Dans cet exemple, deux processus sont démarrés et partagent un canal. Ce canal sert à l'envoi du message et à l'envoi de la réponse :

```
import stackless

def pong_server(channel):
    while True:
        message = channel.receive()
        if message == "ping":
            channel.send("pong")
        else:
            channel.send(None)

def ping_client(channel):
```

```
        channel.send("ping")
        reponse = channel.receive()
        print "Réponse", reponse

ch=stackless.channel()

stackless.tasklet(pong_server)(ch)
stackless.tasklet(ping_client)(ch)
stackless.run()
```

Un canal de communication est d'abord créé. Ensuite, deux tasklets sont créés et reçoivent en paramètre le canal de communication, pour le partager. Le tasklet *pong_server* boucle de façon infinie pour répondre aux messages qu'il reçoit sur le canal. Si le message *ping* est reçu, le message *pong* est envoyé sur le même canal. Dans le cas de la réception d'un autre message, une réponse *None* est envoyée. De son côté, le tasklet *ping_client* envoie le message *ping* sur le canal et attend ensuite une réponse. Une fois la réponse reçue, elle est affichée. L'appel à *stackless.run* démarre l'exécution des deux tasklets. Le résultat suivant est alors affiché :

```
Réponse: pong
```

Résumé

En résumé, les tasklets et les canaux de communication de Stackless Python sont un ajout intéressant par rapport à Python. Ils offrent une nouvelle façon de définir des tâches concurrente, tout en gardant une compatibilité complète avec l'implémentation de base de Python.

Contrairement aux autres langages et extensions de langages vus jusqu'à présent, Stackless Python n'offre pas de méthodes pour démarrer des processus à distance ou les migrer. Aucun moyen de communication n'est défini entre plusieurs interpréteurs pour répondre à ces besoins. La possibilité de sauvegarder l'état d'exécution des tasklets peut tout de même offrir une bonne base pour la mise en place d'outils distribués. Pour cette raison, Stackless Python a été retenu pour la réalisation du travail présenté dans ce mémoire.

L'objectif de ce travail est d'étendre Stackless Python pour y introduire des notions de programmation distribuée. Une notion de nœud est introduite pour permettre à plusieurs interpréteurs de communiquer entre eux. Les tasklets peuvent être migrés

d'un nœud à un autre, en cours d'exécution, et les canaux de communication qu'ils utilisent continuent de fonctionner, même pour des communications entre deux nœuds.

Comparativement à Erlang et Termite Scheme, l'utilisation d'un langage objet pour la mise en place de telles notions offre des défis différents à relever. Dans Stackless Python, le partage d'objets est possible entre tasklets et demande une gestion particulière lorsqu'un objet doit être envoyé vers un autre interpréteur. De plus, pour la migration d'un tasklet, tout son environnement mémoire doit être emporté avec lui. Les langages fonctionnels n'ont pas ces problèmes, car le partage d'objet est impossible entre processus.

1.3 Conclusion

Dans ce chapitre, nous avons donc exploré différentes algèbres de processus et différents langages de programmation ou extensions de langages de programmation. Cette exploration a permis de voir plusieurs caractéristiques qu'ils ont tous en commun, mais aussi certaines caractéristiques spécifiques à chacun d'entre eux. Par exemple, certains de ces algèbres, langages de programmation et extensions de langages de programmation supportent la mobilité de code. Cette mobilité peut permettre le démarrage de processus à distance, mais aussi la migration de processus en cours d'exécution. Différents modèles de communication ont aussi été vus : canaux de communication, boîte aux lettres, synchronisation, etc. Les tableaux 1.6 et 1.7 présentent une comparaison des principales caractéristiques rencontrées.

Les outils permettant le parallélisme et le calcul distribué ne se limitent pas à ceux présentés dans ce chapitre. Pour en mentionner quelques-uns d'intérêt, on retrouve les outils suivants :

- MPI [17] (*Message Passing Interface*) est un système d'échange de messages entre processus et de gestion de processus destiné aux systèmes distribués. Ce système est utilisé sur plusieurs des plus gros superordinateurs actuels. Il offre un ensemble de services et d'interfaces de programmation pouvant être utilisés dans plusieurs langages.
- UPC [46] (*Unified Parallel C*) est une extension au langage C permettant d'exécuter en parallèle les itérations d'une boucle. Cette extension est particulièrement adaptée aux ordinateurs composés de plusieurs processeurs.
- TOP-C [44] offre une interface de programmation permettant de transformer un programme écrit en langage C en un programme distribué avec peu de modifications. Un programme maître est démarré sur un ordinateur et celui-ci s'occupe de

démarrer des copies esclaves du programme, sur des ordinateurs à distances, pour exécuter des tâches précises. L'échange de messages entre le maître et les esclaves se fait à l'aide de MPI.

Bien qu'il existe une très grande quantité d'outils permettant la création de programmes parallèles et distribués, plusieurs d'entre eux n'offrent qu'un ensemble très limité de fonctionnalités. On retrouve par exemple la parallélisation de boucles, les systèmes d'échange de messages entre processus, les outils automatisés de copie et de démarrage d'un programme sur un ordinateur distant, les mémoires distribuées, etc. La plupart de ces outils n'offrent pas la mobilité partielle de code qui est nécessaire au principal objectif de ce mémoire, soit la migration de processus.

Dans le prochain chapitre, nous présentons un modèle de traitement distribué adapté aux langages de programmation objets. Ce modèle reprend la plupart des caractéristiques vues dans le présent chapitre, mais présente aussi certaines contraintes rencontrées avec ce type de langage. Ce modèle mène à l'implémentation d'une extension de Stackless Python permettant la création de programmes distribués.

Propriétés / Langages	CCS	π-calcul	Calcul ambiant
Communication	Canaux	Canaux	Agent (Ambiants mobiles)
Synchronicité	S	S	A
Démarrage de processus distant	N	O avec extension	O
Clonage de processus	N	N	O
Mise à jour de code	N	N	N
Parallélisme / Concurrence	C	C	C
Mise en parallèle	O	O	O
Échange de canaux	N	O	Pas de canaux
Migration de processus (via communication)	N	N	Migration d'ambiant

TABLE 1.6 – Comparaison des algèbres de processus.

Propriétés / Langages	Erlang	Termite Scheme	JCSP.net	Stackless Python	JoCaml
Communication	Boîte aux lettres	Boîte aux lettres	Canaux	Canaux	Canaux fonctions
Synchronicité	A	A	S	S	A/S
Démarrage de processus distant	O	O	O	N	O
Clonage de processus	N	O	O	O	N
Mise à jour de code	O	O	N	O	N
Parallélisme / Concurrence	P	C	P	C	C
Mise en parallèle	O	O	O	O	O
Échange de canaux	O	O	O	O	O
Migration de processus (via communication)	N	O	N	En quelque sorte	O

TABLE 1.7 – Comparaison des langages de programmation ou extensions de langages.

Chapitre 2

Modèle de traitement concurrent et distribué

En lien avec les algèbres et les langages ou extensions de langages qui ont été présentés précédemment, ce chapitre reprend et complète les principaux concepts nécessaires à la mise en place d'outils permettant de créer un environnement distribué basé sur la migration de processus. Ces concepts sont maintenant présentés dans le contexte des langages à mémoire partagée, qui couvre généralement les langages impératifs et objets. Pour permettre de bien comprendre comment chacun des concepts s'assemble avec les autres pour la construction d'un système complet, le chapitre présente les concepts de la façon suivante.

- Une introduction sur les processus permet tout d'abord d'explorer les différents types de processus et la façon dont ils se comportent. La migration de ceux-ci d'un système à un autre, pendant qu'ils sont en cours d'exécution, demande certaines conditions. Ces conditions sont exprimées dans le cadre d'un système distribué hétérogène pouvant être composé de différents systèmes d'exploitation et architectures matérielles.

- Ensuite, le concept de nœud est présenté. Un nœud est un élément d'un système distribué qui contient des processus ainsi que leurs environnements mémoires. Chacun des nœuds est en mesure de communiquer avec les autres pour l'échange de données ou la migration de processus.

- Pour qu'un traitement puisse être divisé en plusieurs processus, il doit exister un moyen de communication entre ceux-ci pour l'échange de données. Des canaux de communication synchrones et distribués permettent ces échanges et gèrent de

façon transparente la migration de processus d'un nœud à un autre.

– Le déplacement d'un processus d'un nœud à un autre implique une copie de son code. De plus, ce code peut avoir des dépendances avec certaines bibliothèques ou données. Un système automatisé de récupération des dépendances doit être mis en place pour qu'un processus migré puisse récupérer ce dont il a besoin pour continuer son exécution.

– Un environnement dans lequel des objets peuvent être partagés entre plusieurs processus impose certaines contraintes quant à la migration d'une partie des objets. Dans certains cas, les objets doivent être copiés et dans d'autres, ils sont trop volumineux et on doit y accéder à distance. Ces différents cas sont explorés et des outils sont proposés pour offrir toute la souplesse nécessaire selon les besoins du programmeur.

– La plupart des programmes ont besoin d'utiliser les méthodes d'entrée et de sortie des systèmes d'exploitation, comme un clavier pour la saisie de données et le terminal virtuel pour l'affichage. Ces méthodes sont malheureusement liées au système sur lequel un processus s'exécute et ce dernier ne peut les déplacer avec lui lorsqu'il est migré vers un autre système. Un système de redirection des entrées et des sorties est présenté, permettant à un processus de continuer d'utiliser les méthodes d'entrée et de sortie spécifiques à un système d'exploitation suite à une migration.

– Finalement, dans un système distribué, certains processus sont dépendants d'autres processus pour l'obtention du résultat de certains traitements. Un système de relation entre processus et de lancement d'exception doit exister pour informer un processus d'une erreur survenue à l'intérieur d'un autre processus avec qui il est en relation. Cette relation est optionnelle et peut être définie selon les besoins.

2.1 Processus

Un programmeur qui veut introduire la notion de parallélisme dans un programme, dans le but de séparer les tâches qui peuvent être exécutées en même temps, doit se familiariser avec la duplication de processus et avec les fils d'exécution. L'utilisation de ces deux concepts lui permet de diviser un programme en plusieurs parties distinctes du point de vue du système d'exploitation. Ces processus peuvent par ailleurs communiquer très facilement entre eux et même partager des espaces mémoire. Étant

donné qu'il les gère comme des processus distincts, le système d'exploitation peut alors exécuter plusieurs d'entre eux simultanément, si l'ordinateur hôte dispose de plusieurs processeurs ou d'un processeur multicœur.

2.1.1 Processus et duplication

Quoique très près l'un de l'autre, la duplication de processus et les fils d'exécution ont des différences majeures. Le principe de duplication de processus a été introduit avec les systèmes d'exploitation UNIX. Son fonctionnement est simple. Le processus effectue un appel à la fonction POSIX *fork* [41]. Le système d'exploitation effectue alors une copie de l'état d'exécution du programme ainsi que de sa mémoire.

Cette copie devient un processus à part entière avec son propre identifiant de processus. Au niveau du système d'exploitation, elle est aussi identifiée comme un enfant du processus qui a été copié. L'appel à la fonction *fork* retourne un résultat dans les deux cas et c'est avec la valeur de retour que l'on peut discerner le parent de l'enfant. La valeur est 0 pour l'enfant et elle correspond à l'identifiant unique du processus enfant pour le parent. C'est en vérifiant cette valeur que le programmeur peut décider quoi faire pour la suite du programme.

Étant donné qu'une copie complète du processus est faite, aucune mémoire n'est partagée. D'autres moyens doivent alors être mis en place pour que le parent et l'enfant puissent échanger des données. Pour répondre à ce besoin, des systèmes de communication interprocessus ont été mis en place au niveau des systèmes d'exploitation. Le plus simple est le tube (*pipe*) [31] des systèmes UNIX.

Un tube est un canal de communication anonyme avec un bout en lecture seule et un autre bout en écriture seule. Chacun de ces bouts est créé à l'aide de la fonction *pipe* [42] et est présenté comme un descripteur de fichier au programmeur. La lecture et l'écriture sur chacun des bouts se font donc de la même façon qu'avec les fichiers. Pour qu'un tube soit utilisé entre deux copies d'un processus, il doit être créé par le processus qui sera dupliqué. Une fois la duplication complétée, chacune des copies du processus a une copie des descripteurs de fichiers, qui pointent vers un unique tube. Pour éviter le blocage qui surviendrait si les deux copies du processus tentaient de lire en même temps sur le tube, le descripteur de fichier qui n'est pas utilisé, soit celui en lecture seule ou celui en écriture seule, devrait être fermé par chacun des processus qui utilisent le tube. La communication peut alors commencer.

2.1.2 Fils d'exécution

Les fils d'exécution [13] sont un autre moyen couramment utilisé pour introduire le parallélisme à l'intérieur d'un programme. Ce sont des processus légers pouvant être démarrés à l'intérieur d'un unique processus (lourd). Par conséquent, les fils d'exécution ne sont pas des processus à part entière, mais ils sont tout de même gérés par l'ordonnanceur du système d'exploitation [37]. Celui-ci peut alors en exécuter plusieurs en même temps lorsque le système dispose d'un processeur multicœur.

À l'intérieur d'un processus, plusieurs fils d'exécution partagent donc le même code, le même espace mémoire et les mêmes ressources. Chacun des fils d'exécution détient tout de même son propre contexte, composé d'un pointeur d'instruction, de registres et d'une pile. Ce contexte est plus petit que celui d'un processus complet. Au niveau du processeur, le changement de contexte nécessaire à l'ordonnancement de plusieurs fils d'exécution faisant partie d'un même processus est alors plus rapide que le changement de contexte complet nécessaire à l'ordonnancement de processus. C'est pour cette raison qu'un fil d'exécution est considéré comme un processus léger.

Un espace mémoire partagé a tout de même certains inconvénients. Lorsqu'il est en cours d'exécution, un fil demeure dans cet état (actif) pour un certain nombre d'instructions avant que l'ordonnanceur le bloque et donne la main à un autre processus ou fil d'exécution pour qu'il puisse être exécuté. Chacun des processus et fils d'exécution est ordonnancé à tour de rôle, ce qui permet de partager un unique processeur. Avec un espace mémoire partagé, il est possible qu'un fil d'exécution A modifie une zone mémoire et doive la lire à nouveau quelques instructions plus loin. Comme l'exécution de A peut être interrompue avant que la zone mémoire ne soit lue à nouveau, un fil B peut alors modifier la même zone mémoire pour lui attribuer une valeur qui n'est plus cohérente pour A. Lorsque le fil A est de nouveau exécuté, il lit alors une valeur erronée.

Pour éviter ce type de problème, un système de verrous doit être mis en place. Un fil d'exécution peut alors demander un accès exclusif à une zone mémoire pour le temps durant lequel il en a besoin. Un autre fil d'exécution qui tente d'utiliser la même zone mémoire sera alors bloqué tant et aussi longtemps que le fil détenant le verrou ne le relâche pas. Ce mécanisme permet donc la complétion d'une opération en s'assurant de la validité de toutes les données dont elle a besoin.

Le système de verrous a tout de même un désavantage. Si deux fils d'exécution détiennent des verrous sur des zones mémoires et que chacun d'entre eux veut accéder à une zone mémoire verrouillée par l'autre, ils seront alors tous les deux bloqués

indéfiniment. Cet événement se nomme interblocage [51]. Dans la plupart des cas, un interblocage implique le redémarrage complet du programme à moins qu'un mécanisme n'ait été mis en place par le programmeur pour interrompre les fils d'exécutions bloqués. De plus, l'interblocage indique habituellement une erreur de programmation devant être corrigée.

Beaucoup de ces interblocages surviennent de façon aléatoire et sont très difficiles à identifier. Lors d'exécutions différentes d'un même programme, qui contient plusieurs fils d'exécutions, l'exécution des fils ne se fait pas nécessairement à la même vitesse, ni dans le même ordre. Ceci est dû aux algorithmes d'ordonnancement complexes des systèmes d'exploitation qui tiennent compte de plusieurs paramètres comme la charge du système, le nombre de processus et de fils d'exécution en file d'attente ainsi que la priorité de chacun d'eux. Un programmeur qui utilise les fils d'exécution doit donc toujours être attentif à ce risque lorsqu'il doit mettre en place des verrous.

Les problèmes mentionnés font des fils d'exécution une technique souvent difficile à utiliser. Il ne faut d'ailleurs pas en abuser, car un trop grand nombre de fils d'exécution peut surcharger l'ordonnanceur. Le système d'exploitation passe alors plus de temps à ordonnancer les fils qu'à les exécuter. Il existe des fils d'exécution encore plus légers qui permettent de résoudre certains problèmes et qui offrent d'autres avantages.

2.1.3 Continuations

Plusieurs techniques de concurrence, encore plus légères que les fils d'exécution, ont été mises à la disposition des programmeurs par les langages de programmation. Elles permettent de bien définir les différentes tâches qu'un programme a à accomplir et donnent un plus grand contrôle sur son flux d'exécution. Dans la plupart des cas, ces techniques dont dites collaboratives. Les tâches ne sont donc pas exécutées de façon parallèle. Une tâche collaborative doit explicitement donner la main à une autre pour qu'elle soit exécutée à son tour.

À la base des techniques collaboratives, nous retrouvons les continuations. Elles sont souvent utilisées pour la construction de méthodes plus évoluées telles que les coroutines. Une continuation peut être décrite comme suit : pour chacune des instructions d'un programme, une continuation est une fonction représentant le reste des instructions. Prenons par exemple la suite d'instructions suivante :

x=3
y=2

```
print 2+3
```

Lorsque l'instruction **x=3** est exécutée, la continuation à ce point du code correspond à la séquence d'instructions **y=2** et **print 2+3**. Il en va de même lorsque **y=2** est exécutée. Elle aura comme continuation l'instruction **print 2+3**.

D'un point de vue plus pratique, nous retrouvons les continuations de première classe. Ce sont des outils offerts par certains langages de programmation, qui permettent de manipuler le flux d'exécution d'un programme. Elles offrent la possibilité de sauvegarder l'état d'exécution à un point donné d'un programme. La dernière instruction exécutée, ainsi que les valeurs des variables locales, sont sauvegardées dans un objet *continuation* que le programme peut manipuler.

Une continuation est dite « de première classe » lorsqu'elle peut être utilisée comme tout autre objet d'un langage. Elle peut alors être passée en paramètre lors de l'appel d'une fonction ou même sérialisée pour être rechargée plus tard, possiblement sur un autre système. Un appel de cette continuation permet de récupérer l'état d'exécution sauvegardé. La pile d'appels ainsi que les variables locales sont rechargées et l'exécution du programme saute au point où la continuation avait été créée.

Un exemple intéressant de ce à quoi une continuation ressemblerait dans le monde réel a été énoncé par *Luke Palmer* en 2004 [19] :

> « Disons que vous vous retrouvez dans la cuisine, devant le réfrigérateur, en pensant à un sandwich. Vous prenez une continuation et vous la mettez dans votre poche. Ensuite, vous sortez un peu de dinde et du pain du réfrigérateur et vous faites votre sandwich, qui se retrouve maintenant sur le comptoir. Vous appelez maintenant la continuation, qui se trouve dans votre poche, et vous vous retrouvez à nouveau devant le réfrigérateur, en pensant à un sandwich. Heureusement, il y a maintenant un sandwich sur le comptoir et tout le matériel utilisé pour le faire a disparu. Vous le mangez donc. :-) »

Un appel à une continuation peut habituellement recevoir un paramètre. La valeur attribuée à ce paramètre est retournée au point où l'état d'exécution est récupéré. Elle permet de différencier un état d'exécution rechargé de l'exécution initiale.

Voici un exemple simple qui démontre la création d'une boucle à l'aide d'une continuation :

```
def boucle(n):
    c = continuation()
    k = c.create(n)
    if k:
        afficherEtContinuer(c, k)

def afficherEtContinuer(c, k):
    print "k vaut:", k
    c.call(k-1)
    print "Fin de l'affichage"

boucle(5)
```

La première étape consiste à créer un objet qui contient la continuation. À l'intérieur de la fonction *boucle*, la continuation est effectivement créée lors de l'appel à *c.create*. L'état d'exécution est désormais sauvegardé. La valeur passée à *c.create* sera retournée immédiatement pour être assignée à *k*.

Un appel est par la suite effectué à *afficherEtContinuer* pour afficher cette valeur et appeler la continuation avec la valeur *k-1*. Le flux d'exécution du programme saute alors directement à l'endroit où la continuation avait été créée, soit lors du premier appel à *c.create*. Cet appel se termine à nouveau et retourne la valeur passée à *c.call*.

L'appel précédent de la fonction *afficherEtContinuer* ne retourne jamais et est effacé de la pile d'appels comme si rien ne s'était passé. La ligne **print "Fin de l'affichage"** n'est alors jamais exécutée étant donné que le flux d'exécution du programme est toujours modifié tout juste avant cette instruction.

Lorsque la valeur de k vaut 0, la fonction *afficherEtContinuer* n'est plus appelée et la fonction *boucle* termine. Le programme termine alors. Une exécution de ce programme affiche ce qui suit :

```
k vaut: 5
k vaut: 4
k vaut: 3
k vaut: 2
k vaut: 1
```

Les continuations sont donc très simples à utiliser et peuvent offrir une très grande puissance pour la résolution de certains problèmes qui seraient très difficiles sans elles.

Par contre, comme pour l'instruction *goto*, un abus d'utilisation peut introduire de la complexité dans la structure d'un programme, qui devient alors difficile à comprendre.

2.1.4 Coroutines

Les continuations permettent l'introduction de techniques de concurrence basées sur celles-ci. La première est la coroutine. Une coroutine est une tâche pouvant être exécutée en concurrence avec d'autres coroutines, à l'intérieur d'un même programme. Elles fonctionnement de façon coopérative, c'est-à-dire qu'une coroutine doit donner elle-même la main à une autre coroutine pour que cette dernière soit exécutée. Cette technique n'offre donc pas un réel parallélisme lorsqu'un processeur multicœur est disponible.

Le fonctionnement des coroutines est simple. Une fonction doit d'abord être définie comme point d'entrée de la coroutine. Un objet *coroutine* est ensuite créé et la fonction qui sera appelée est liée à cet objet. Ces étapes sont répétées pour chacune des coroutines à créer. Une fois le tout en place, une méthode *switch* est appelée sur la première coroutine pour la lancer.

Lorsqu'une coroutine veut interrompre son exécution, elle peut appeler la méthode *switch* d'une autre coroutine pour lui donner la main. Elle reste alors bloquée tant et aussi longtemps que sa propre méthode *switch* ne se fait pas appeler à nouveau. Un programme composé de coroutines termine lorsqu'une d'entre elles termine sans avoir donné la main. Voici un exemple :

```
def boucle():
    while k:
        coro_afficheur.switch()
        k = k - 1

def afficheur():
    while True:
        print "k vaut:", k
        coro_boucle.switch()

coro_boucle = coroutine()
coro_afficheur = coroutine()

coro_boucle.bind(boucle)
coro_afficheur.bind(afficheur)
```

```
k = 5

coro_boucle.switch()
```

Dans cet exemple, une coroutine est créée pour incrémenter la valeur d'une variable et une autre coroutine est créée pour afficher cette valeur. Une valeur initiale est ensuite assignée à *k* et finalement, la main est donnée à la coroutine *coro_boucle* lorsque la méthode *coro_boucle.switch* est appelée. Cette coroutine décrémente la valeur de la variable *k* tant qu'elle est supérieure à 0. À chaque itération de sa boucle, elle donne la main à la coroutine *coro_afficheur*.

De son côté, la coroutine *coro_afficheur* boucle indéfiniment pour afficher la valeur actuelle de *k*. Une fois la valeur affichée, elle donne à nouveau la main à *coro_boucle*. Lorsque l'exécution de la fonction *boucle* termine et que cette dernière retourne, le programme termine à son tour. Malgré le fait que la fonction *afficheur* soit exécutée dans une coroutine séparée, elle ne fait pas partie du flux d'exécution du programme au moment où la fonction *boucle* retourne.

Bien évidemment, le résultat de l'exécution est le même que l'exemple sur les continuations :

```
k vaut: 5
k vaut: 4
k vaut: 3
k vaut: 2
k vaut: 1
```

Les continuations se prêtent bien à l'implémentation des coroutines. Chaque coroutine en attente d'être exécutée se voit associer une continuation. Lorsqu'une coroutine donne la main, une continuation est créée à ce moment, pour sauvegarder son état d'exécution. Un appel est ensuite fait à la continuation de la coroutine vers laquelle le flux d'exécution doit être redirigé. Ce processus est contenu à l'intérieur de la méthode *switch* d'une coroutine, ce qui masque le concept de continuation.

En comparaison avec les processus et les fils d'exécution, le saut d'une coroutine à une autre se fait alors beaucoup plus rapidement, car il n'est pas nécessaire d'effectuer un changement de contexte au niveau du processeur. En terme de temps d'exécution, l'appel d'une continuation équivaut sensiblement à l'appel d'une fonction. Plusieurs sauts peuvent alors être effectués d'une coroutine à une autre pendant une période

d'exécution du processus qui les contient, période d'exécution attribuée par l'ordonnanceur du système d'exploitation.

Comme les coroutines sont exécutées à l'intérieur d'un seul processus ou fil d'exécution, elles ne peuvent être exécutées en parallèle. Un cas particulier des coroutines se nomme *fibre* [50] et est souvent offert par le système d'exploitation. Celui-ci est alors capable de gérer une forme de parallélisme à l'aide de fils d'exécution. Toute fibre arrêtée peut être relancée à l'intérieur de n'importe quel fil d'exécution disponible.

Comme les fibres sont très légères par rapport aux fils d'exécution, il peut être avantageux de séparer les tâches pouvant être exécutées en parallèle avec un très grand nombre de fibres, exécutées à l'intérieur d'un nombre restreint de fils d'exécutions. En fait, il ne devrait pas y avoir plus de fils d'exécutions que le nombre de processeurs ou de cœurs disponibles dans le système. C'est ce qu'on nomme « modèle de fils de type M:N ». Les changements de contexte sont donc diminués au niveau du processeur et on peut sauter rapidement d'une fibre à une autre lorsque nécessaire.

Le concept de coroutine commence à se rapprocher des modèles de processus et de fils d'exécution offerts par les systèmes d'exploitation. Il est possible, en utilisant des coroutines, de séparer les tâches distinctes d'un programme, tout en évitant la surcharge de l'ordonnanceur du système d'exploitation que peut engendrer l'utilisation des fils d'exécution. Mais cela demande un certain travail au programmeur qui doit spécifier à qui chaque coroutine donne la main.

2.1.5 Microprocessus

La coroutine est un concept de bas niveau, auquel il faut indiquer à qui passer la main. Il est encore possible de simplifier son utilisation. Un système de microprocessus coopératifs qui ressemble aux processus ou fils d'exécution du système d'exploitation peut facilement être implémenté à l'aide de coroutines. Il suffit d'ajouter une file d'attente qui contient les références vers tous les microprocessus à exécuter.

Pour qu'un microprocessus donne la main à un autre, il n'a pas besoin de le connaître. Une fonction générique faisant appel à un microordonnanceur est appelée. Ce dernier s'occupe alors de prendre le premier microprocessus en file d'attente et de rediriger le flux d'exécution vers celui-ci. Le microprocessus qui donne la main est alors placé à la fin de la file d'attente.

Le programmeur qui veut séparer des tâches peut alors démarrer plusieurs micro-

processus, qui seront exécutés à tour de rôle. Ce modèle est utilisé par Stackless Python pour ses microprocessus nommés *tasklet* (on parle alors de tâches). Par ailleurs, ces microprocessus ont accès à des canaux de communication pour l'échange de données. Voici l'exemple précédent, implémenté à l'aide de microprocessus :

```
import stackless

def boucle(c, k):
    while k:
        c.send(k)
        k = k - 1

def afficheur(c):
    while True:
        k = c.receive()
        print "k vaut:", k

c = stackless.channel()
stackless.tasklet(boucle)(c, 5)
stackless.tasklet(afficheur)(c)

stackless.run()
```

Dans cet exemple, un canal de communication est préalablement créé. Il sert à l'échange de la valeur de k entre les deux microprocessus. Les deux microprocessus sont ensuite créés et reçoivent le canal de communication en paramètre (ce canal devient partagé entre les deux microprocessus). En plus du canal de communication, le microprocessus *boucle* reçoit aussi la valeur initiale de k. Suite à sa création, chaque microprocessus est placé dans la file d'attente d'un ordonnanceur pour être exécuté.

L'ordonnanceur est démarré lors de l'appel à *stackless.run*. Il s'occupe d'exécuter à tour de rôle les microprocessus présents dans sa file d'attente. Chacun des microprocessus est exécuté tant et aussi longtemps qu'il ne fait pas un appel à l'ordonnanceur, pour donner la main au prochain microprocessus de la file d'attente. Avec Stackless Python, la fonction *stackless.schedule* est utilisée pour faire cette action.

Compte tenu de l'utilisation d'un canal, les microprocessus n'ont pas à donner explicitement la main. Les canaux sont bloquants et lorsqu'une donnée ne peut être envoyée ou reçue immédiatement, le microprocessus est retiré de la file d'attente et la main est donnée immédiatement au prochain microprocessus présent dans la file d'attente. Le microprocessus bloqué est placé dans une file d'attente sur le canal et sera débloqué

lorsque la communication pourra être complétée. Dans cet exemple, l'exécution saute donc d'un microprocessus à l'autre lors des requêtes au canal.

Lorsque le microprocessus *boucle* termine, le microprocessus *afficheur* bloque sur le canal et est retiré de la file d'attente de l'ordonnanceur. Étant donné qu'il ne reste aucun autre microprocessus en file d'attente, l'ordonnanceur termine le programme, car le microprocessus *afficheur* est bloqué à jamais. La boucle infinie ne cause donc pas de problème. Encore une fois, le même résultat est affiché à l'écran :

```
k vaut: 5
k vaut: 4
k vaut: 3
k vaut: 2
k vaut: 1
```

Comme ils sont basés sur les coroutines, les microprocessus profitent de certaines caractéristiques de celles-ci et des continuations. La principale différence par rapport aux coroutines est qu'il n'est pas nécessaire que le microprocessus ait une connaissance des autres pour pouvoir donner la main. Un microordonnanceur remplit ce rôle.

2.1.6 Microprocessus et langage interprété

Avec un langage interprété, l'interpréteur détient un très grand contrôle sur l'exécution d'un programme, ainsi que sur sa mémoire. Ce contrôle offre d'énormes possibilités pour l'implémentation de microprocessus basés sur les coroutines, elles-mêmes basées sur les continuations.

Pour avoir ce contrôle, un interpréteur doit avoir sa propre représentation du code et des objets d'un programme. Même à la compilation, certains langages interprétés tels que Java et Python, produisent un code binaire intermédiaire (*bytecode*) qui est indépendant de toute architecture matérielle et logicielle. Avec une représentation propre à l'interpréteur, la majorité des spécificités de chacune des architectures supportées sont alors masquées.

Le contrôle de l'exécution d'un programme a aussi une importance dans l'implémentation des continuations. Lors de la création d'une continuation, l'interpréteur doit sauvegarder une copie de la pile d'appels, ainsi que des variables locales de chacune des fonctions et méthodes appelées. Cette copie, normalement utilisée pour transférer le flux d'exécution au point sauvegardé, peut aussi être sérialisée. Il n'est pas nécessaire

de recharger la copie sérialisée sur la même instance de l'interpréteur. Cette possibilité ouvre la porte à la migration de microprocessus d'un ordinateur à un autre, en cours d'exécution.

2.1.7 Migration de microprocessus

Pour le programmeur, le fait de pouvoir déplacer des microprocessus d'un interpréteur à un autre offre plusieurs possibilités. Il devient très simple de mettre en place un système de nœuds distribués avec peu d'installation et de configuration logicielle. Dans un tel système, chaque nœud est un interpréteur pouvant communiquer avec d'autres nœuds pour recevoir des microprocessus et les exécuter. Avec cette structure, plusieurs problèmes relatifs au traitement distribué peuvent être réglés. En voici quelques exemples :

- Clients légers qui récupèrent des parties de programmes s'exécutant sur un serveur.
- Création d'agents mobiles pouvant être déplacés d'un système à un autre sans que le programmeur n'ait à gérer la sauvegarde et la récupération de leurs états.
- Déplacement de traitements d'un système à un autre pour qu'un administrateur puisse effectuer un entretien sur le système où ils étaient initialement exécutés.

La migration de microprocessus est par contre difficile à réaliser avec les langages à mémoire partagée. Le programmeur a souvent toute la souplesse nécessaire pour structurer ses programmes comme bon lui semble, mais cette souplesse apporte son lot de problèmes. Par exemple, lorsque deux microprocessus partagent un même espace mémoire, comment déplacer l'un d'entre eux vers un autre nœud qui n'aura pas accès à cet espace ?

Un problème semblable se pose avec certaines ressources du système ne pouvant être déplacées comme les fichiers, le clavier, l'écran et les connexions réseau. Si le microprocessus utilise ces ressources, comment gérer l'accès à celles-ci lorsqu'il est migré ?

Malgré les avantages de la migration de microprocessus, il est difficile de définir un modèle qui gère tous les problèmes potentiels. Certaines limites doivent être imposées au programmeur pour rendre cette tâche possible. Des outils doivent tout de même lui être offerts pour gérer les cas qui pourraient empêcher la migration d'un microprocessus ou créer des effets de bord (comportements inattendus). Les méthodes employées pour résoudre ces problèmes ainsi que les outils offerts au programmeur font l'objet des prochaines sections.

2.2 Nœuds

De façon générale en mathématique et en informatique, un nœud est un point d'intersection dans un graphe [4]. Dans un environnement distribué, un nœud est une entité capable d'effectuer un traitement ou un calcul et il est connecté à d'autres nœuds afin d'échanger des données. Le tout forme un réseau qui est une forme de graphe. Avec l'introduction de la migration de microprocessus, le terme *nœud* est donc utilisé pour représenter une instance de l'interpréteur d'un langage de programmation, capable d'exécuter des microprocessus.

Le nœud a donc comme rôle premier de gérer tous les services offerts par un langage de programmation. Il met d'abord à la disposition du programmeur tous les outils nécessaires pour la construction de ses programmes (langage et bibliothèques) et leurs exécutions (interpréteur). Ensuite, lorsqu'il exécute un programme, le nœud s'occupe des tâches suivantes :

- La création et l'exécution de chacun des microprocessus du programme.
- La gestion des outils permettant aux microprocessus de communiquer entre eux.
- La gestion de la mémoire et de son nettoyage lorsque des objets ne sont plus utilisés.
- L'accès aux ressources du système à l'intérieur duquel il se trouve.

Dans un contexte distribué, le deuxième rôle du nœud est de gérer sa relation avec les autres nœuds. Deux nœuds vont entrer en relation lorsqu'une connexion est établie entre eux. Cette connexion est nécessaire, par exemple, pour l'utilisation d'un canal de communication entre deux microprocessus distants ou pour l'utilisation de ressources à distance.

Cette connexion permet donc à deux nœuds de communiquer. Différents types d'échanges sont possibles. Un protocole de communication est établi pour chacun de ces types. L'ensemble de ces protocoles correspond donc à un langage de communication compris par l'ensemble des nœuds d'un environnement distribué ; le programmeur n'a pas à s'en préoccuper.

Pour que plusieurs nœuds puissent entrer en relation, chaque nœud doit être identifié de façon unique. Cet identifiant doit permettre à un nœud de retrouver par un réseau de communication un autre nœud avec lequel il désire communiquer. L'identifiant est utilisé lors d'une demande de migration d'un microprocessus pour indiquer le nœud de destination de l'opération. C'est suite à la première migration de microprocessus que tous les types d'échanges possibles entre deux nœuds peuvent survenir.

Dans un contexte d'utilisation d'un réseau *TCP/IP*, pour connecter les nœuds entre eux, il est approprié d'utiliser un identifiant unique composé d'une adresse *IP* et d'un port *TCP*. L'adresse et le port utilisés pour l'identification d'un nœud doivent permettre à tous les autres nœuds présents sur le réseau de le rejoindre. Dès qu'un nœud est identifié, il écoute alors sur le port *TCP* faisant partie de l'identifiant pour être en mesure de recevoir les requêtes en provenance d'autres nœuds.

2.3 Canaux

2.3.1 Méthodes de communication interprocessus

Avec un programme composé de plusieurs tâches parallèles, il devient évident qu'un moyen de communication est nécessaire pour que chaque tâche puisse échanger les données requises pour son accomplissement. Il a déjà été mentionné que l'utilisation d'une mémoire partagée avec les fils d'exécution et des microprocessus apporte son lot de problèmes. Deux fils utilisant une même zone mémoire peuvent créer des effets de bord et l'utilisation de verrous introduit un risque d'interblocage.

Lorsque les microprocessus utilisant une mémoire partagée sont exécutés sur un seul processus du système d'exploitation, la majorité de ces problèmes peut être résolue à cause même de la nature coopérative de ces microprocessus. Un microprocessus qui exécute une tâche en utilisant une zone mémoire partagée a le pouvoir de la terminer complètement avant de donner la main à un autre microprocessus. L'intégrité de la zone mémoire partagée est alors assurée et les verrous deviennent superflus.

Le modèle de microprocessus avec mémoire partagée a tout de même ses limites. Lorsque l'on veut introduire le concept de migration de microprocessus, il faut pouvoir copier le contexte associé à un microprocessus d'un nœud à un autre en évitant le plus possible les effets de bord. La mémoire partagée fait partie du contexte d'un microprocessus à migrer, mais elle fait aussi partie des contextes des autres microprocessus l'utilisant. Lorsqu'un microprocessus est migré, il n'a plus accès à cette mémoire partagée, car elle se trouve dans le nœud où il était initialement exécuté. Nous avons donc le choix de copier cette mémoire, de la rendre accessible à distance ou tout simplement d'empêcher l'utilisation d'une telle mémoire partagée.

Si on se base sur la communication par tube (voir prochaine section) qui est la plus simple à mettre en place avec le modèle de parallélisme par duplication de processus

de UNIX, empêcher l'utilisation d'une mémoire partagée peut sembler une solution intéressante. Ce modèle est connu depuis plusieurs années par les programmeurs et il est très simple à mettre en place. L'idée est donc d'isoler chacun des microprocessus dans son propre environnement, avec sa propre mémoire globale, et d'utiliser des canaux de communication partagés entre microprocessus pour échanger les données.

La mémoire globale reste disponible pour les fonctions et objets associés à un seul microprocessus et les canaux de communication sont utilisés pour que les échanges soient possibles entre plusieurs microprocessus. L'isolation d'un microprocessus dans son propre environnement simplifie aussi la migration, car cet environnement peut être entièrement migré en même temps que le microprocessus, sans causer d'effets de bord sur les autres.

2.3.2 Fonctionnement des tubes

Avant de définir le comportement des canaux de communication, jetons tout d'abord un regard sur celui des tubes UNIX [31]. Les tubes font partie des méthodes de communication interprocessus [40] [18] offertes par les systèmes d'exploitation et sont créés à l'aide de la fonction POSIX *pipe* [42].

Une fois le tube créé la fonction *pipe* retourne deux pointeurs, respectivement vers son bout en lecture et vers son bout en écriture. Ces pointeurs sont utilisés pour l'échange de données par le tube, sous forme de tableaux d'octets. Un tube peut donc être vu comme une liste d'octets qui est remplie avec des données lors de l'écriture et qui est vidée lors de la lecture.

Cette liste a une taille finie et peut se remplir complètement si elle n'est pas vidée par un processus lecteur. Habituellement un processus écrit sur le tube et un ou plusieurs processus lisent ce qui a été écrit. Les fonctions POSIX *write* et *read* sont utilisées pour remplir ces rôles. L'écriture se fait de façon non bloquante et séquentielle, tant qu'il y a de l'espace disponible sur le tube. S'il se remplit, le processus qui veut écrire reste bloqué tant que les données écrites précédemment n'ont pas été lues. De son côté, la lecture reste bloquante tant et aussi longtemps qu'il n'y a pas de données disponibles sur le tube. La lecture se fait aussi de façon séquentielle.

Le fonctionnement de base des tubes est donc synchrone, car il est possible que les opérations de lecture et d'écriture bloquent en tout temps. Les tubes peuvent tout de même être utilisés de façon asynchrone à l'aide d'une méthode non bloquante, qui indique s'il est possible de lire ou d'écrire sur un tube avant que l'opération bloquante

ne soit lancée. Voici un exemple simple, en langage C, de l'utilisation des tubes dans un contexte de duplication de processus :

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main() {
        int tube[2];
        char* message;
        char tampon[15];

        pipe(tube);

        if (fork() == 0) {
                close(tube[1]);
                read(tube[0], tampon, 15);

                printf("Je suis l'enfant!\n");
                printf("Le parent m'envoie le message " \
                        "suivant: %s\n", tampon);

                close(tube[0]);
        } else {
                close(tube[0]);
                printf("Je suis le parent!\n");

                message = "Bonjour monde!";
                write(tube[1], message, 15);
                close(tube[1]);
        }

        return 0;
}
```

Dans cet exemple un tube est tout d'abord créé lors de l'appel à la fonction *pipe*. Les deux bouts du tube sont placés dans le tableau *tube*. Un appel est ensuite fait à la fonction *fork* pour dupliquer le processus. Une valeur de retour de la fonction *fork* égale à 0 indique qu'on se trouve dans la copie enfant du processus. Dans l'exemple, le corps du bloc *if* est donc exécuté pour cette copie du processus. Lorsque la valeur de retour de *fork* est plus grande que 0, on se trouve dans la copie parent. Le corps du

bloc *else* est alors exécuté.

Une fois la duplication complétée, chacune des copies du processus ferme le bout du tube qu'elle n'utilise pas : la copie enfant, qui doit lire sur le tube, ferme le bout en écriture ; la copie parente, qui écrit sur le tube, ferme la copie en lecture. Cette façon de faire évite l'interblocage qui surviendrait dans le cas où les deux copies du processus tenteraient de lire en même temps sur le tube. Une opération faite sur un bout fermé d'un tube génère automatiquement une erreur.

Suite à la fermeture de son bout inutilisé du tube, le parent affiche qui il est et écrit le message « Bonjour monde ! » sur le tube. Une fois le message envoyé, le bout restant du tube est fermé et le parent termine son exécution.

De son côté, l'enfant tente de lire sur le tube. Il reste bloqué tant que le message du parent n'a pas été envoyé. Une fois le message reçu, il affiche qui il est et il affiche ensuite le message qu'il a reçu du parent. Finalement, l'enfant ferme le bout restant du tube et termine son exécution à son tour. Lors de son exécution, cet exemple affiche ce qui suit :

```
Je suis le parent!
Je suis l'enfant!
Le parent m'envoie le message suivant: Bonjour monde!
```

La principale difficulté avec les tubes est qu'ils sont conçus pour échanger le contenu de zones mémoires. Lors de l'appel de la fonction *write*, le programmeur doit fournir l'adresse d'une zone mémoire et le nombre d'octets à écrire dans le tube à partir du début de la zone mémoire fournie. À la lecture, le principe est le même, le programmeur fournit l'adresse d'une zone mémoire tampon et indique le nombre d'octets à lire dans le tube vers cette zone mémoire. La valeur de retour de la fonction *read* indique le nombre d'octets lus.

Ce comportement est normal, car les tubes, tout comme les fichiers et les connexions réseau, sont une méthode de communication de bas niveau pour le système d'exploitation. Il revient au programmeur de définir un protocole de communication pour pouvoir encoder et décoder des données complexes.

L'introduction d'un deuxième processus lecteur complexifie encore plus la situation. Lorsque des données sont écrites sur un tube, il est impossible de savoir quel processus lecteur lira ces données et quelle quantité sera lue. Une partie des données destinées à un processus pourraient alors être lues par un autre processus. Il est alors préférable d'utiliser un tube par processus enfant et même par type de données, pour simplifier

les protocoles de communication.

2.3.3 Canaux de communication

Avec un langage de programmation de haut niveau, il est très simple de reproduire le mode de fonctionnement des tubes tout en corrigeant les faiblesses inhérentes au tube. Une version améliorée des tubes correspond aux canaux de communication. Au même titre que les tubes sont utilisés par les processus, les canaux de communication sont utilisés par les microprocessus. Comme les microprocessus ont un faible coût d'utilisation, nous pouvons aussi en tirer profit pour mettre en place certains comportements reliés aux canaux.

Commençons par la façon d'échanger les données. Nous avons vu que l'utilisation de tableaux d'octets pour lire et écrire sur un tube n'est pas commode pour le programmeur. Dès qu'il veut échanger des données complexes comme des objets, il doit manuellement les encoder et les décoder. La définition d'un protocole de communication s'avère donc nécessaire.

Les tableaux d'octets sont très rarement utilisés avec les langages de programmation de haut niveau. La plupart de ces langages supportent la programmation par objet et offrent des méthodes beaucoup plus simples utilisant les objets. Il est donc évident qu'un canal de communication de plus haut niveau devra transporter des objets. Le programmeur n'a donc pas besoin de se soucier de l'encodage des données et de la taille de ce qui doit être écrit ou lu. Seule la référence vers l'objet à envoyer, référence qui est toujours de taille fixe, est échangée lors de la communication, contrairement aux tubes qui effectuent une copie des données.

Ce comportement est cohérent avec le passage des objets qui se fait par référence dans la plupart des langages objets, lors de l'appel d'une fonction ou d'une méthode. Ce passage par référence fait aussi en sorte que tous les types d'objets, incluant les canaux eux-mêmes, les microprocessus et les objets d'entrées/sorties, peuvent être envoyés via un canal de communication. L'introduction des canaux de communication distribués ajoute certaines restrictions, qui sont expliquées dans les prochaines sections. La figure 2.1 démontre la différence entre la communication par tube et la communication par canal de communication.

Le fait que les canaux de communication envoient les objets par référence assure une atomicité de l'opération. Le programmeur qui utilise un canal de communication est donc certain qu'un objet est complètement envoyé à la fin de l'appel de la méthode

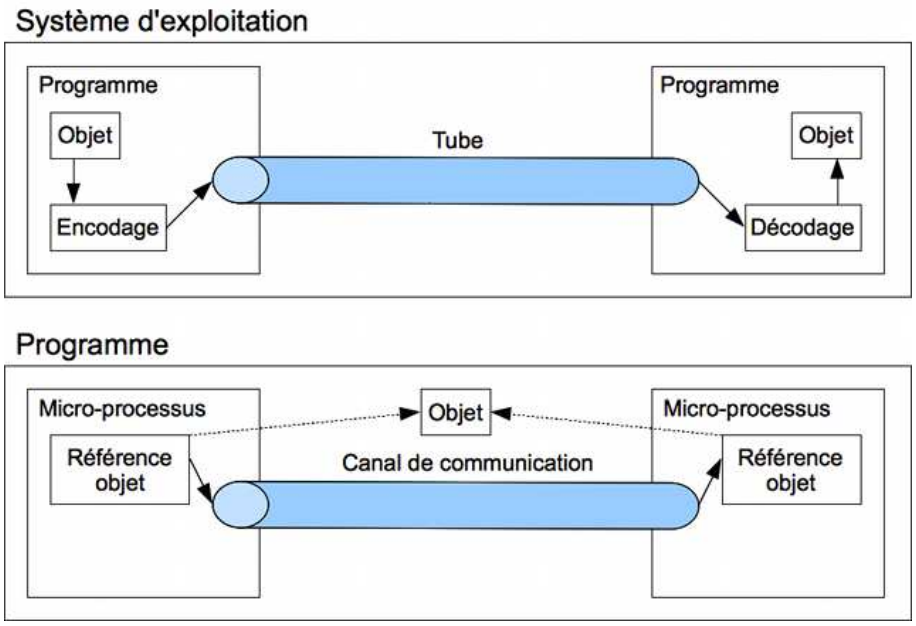

FIGURE 2.1 – Comparaison entre tube et canal de communication.

d'écriture du canal et il est aussi certain que l'objet est complètement reçu lors du retour de la méthode de lecture. Ceci est tout aussi vrai lors de l'envoi de plusieurs objets, de types différents, sur un même canal de communication. Ce comportement est assuré de façon implicite par le passage par référence.

Cette atomicité des opérations sur un canal simplifie beaucoup l'utilisation d'un seul canal de communication par plus d'un microprocessus receveur. Reprenons l'exemple des tubes avec un processus envoyeur et deux receveurs, où il était impossible de savoir quel processus lit une donnée et quelle quantité de données est lue. Les moyens de répondre à ces problèmes sont, soit de toujours effectuer des envois et des réceptions de taille fixe, contenant la totalité des données à échanger, soit d'utiliser un tube pour chaque paire de processus avec l'ajout d'un protocole de communication pour l'encodage et le décodage des données.

Avec des canaux de communication, où seules des références sont échangées, nous sommes assurés qu'une donnée envoyée est reçue complètement et ce, peu importe le nombre de microprocessus envoyeurs ou receveurs sur un canal donné. Un seul microprocessus peut recevoir une donnée présente sur un canal. L'ordre dans lequel les microprocessus reçoivent les objets est l'ordre dans lequel ils ont initié leurs requêtes. Une file d'attente note cet ordre lorsque les requêtes d'envoi ou de réception ne peuvent être immédiatement complétées.

Mode de communication synchrone

Passons maintenant au mode de communication. Il faut tout d'abord se rappeler que les canaux de communications sont utilisés par des microprocessus. Ces microprocessus sont exécutés à l'intérieur d'un nœud, qui s'occupe de les ordonnancer. Dans cette situation, nous privilégions un mode de communication synchrone, pour les canaux de communication, car peu d'effort est nécessaire pour le mettre en place.

En mode synchrone, lorsqu'un microprocessus effectue une demande de lecture ou d'écriture sur un canal de communication, il doit être bloqué si la donnée ne peut être immédiatement échangée avec un autre microprocessus. La façon de bloquer un microprocessus est alors de le retirer de la file d'ordonnancement et de donner immédiatement la main à un autre microprocessus en attente d'être exécuté. Cette façon de faire est la même au niveau du système d'exploitation avec les entrées et sorties bloquantes. Lorsqu'une opération de lecture ou d'écriture effectuée par un processus est impossible sur un tube, le processus est alors retiré de la file d'ordonnancement jusqu'à ce que l'opération soit complétée.

Mode de communication asynchrone

Pour l'implémentation de base de canaux, le mode asynchrone a volontairement été mis de côté. La raison est simple : il est possible de créer des canaux asynchrones à l'aide de canaux synchrones et ce, à faible coût. Pour répondre à ce problème, nous pouvons tirer profit des microprocessus. Lorsqu'un programmeur veut envoyer une donnée de façon asynchrone, il peut alors démarrer un microprocessus temporaire, qui s'occupe d'envoyer la donnée. C'est ce nouveau microprocessus qui reste bloqué sur le canal, en attente de l'envoi de la donnée.

Suite au démarrage du microprocessus temporaire, celui qui a lancé la demande d'envoi asynchrone est immédiatement débloqué. Il peut alors effectuer d'autres traitements et aussi envoyer d'autres messages de façon asynchrones. On retrouve donc un microprocessus, par message en attente d'être envoyé, et chacun de ces microprocessus termine une fois son message envoyé. Pour gérer les problèmes d'envoi, par exemple, lorsqu'un délai maximum d'attente pour l'envoi est atteint, un système événementiel peut être mis en place. Une fonction servant à gérer l'erreur peut alors être appelée.

Pour la réception asynchrone, le principe est le même. Un microprocessus est démarré, pour écouter sur un canal de façon synchrone. Lorsqu'un message est reçu, il

appelle une fonction qui sert à traiter le message. Pendant ce temps, tous les autres microprocessus du nœud peuvent continuer à remplir leurs tâches.

Canal de communication distribué

L'introduction d'un environnement distribué ajoute une certaine complexité et plusieurs contraintes par rapport au comportement des canaux. Dans un tel environnement, plusieurs microprocessus exécutés sur des nœuds différents peuvent partager des canaux de communication. De plus, la migration de microprocessus introduit un possible changement de la topologie entre les différents nœuds.

La première contrainte se situe par rapport à la façon d'échanger les données. Nous avons vu que pour une communication locale à un nœud, seule une référence vers un objet est échangée. Suite à la communication, le même objet est donc utilisé par deux microprocessus. Par contre, suite à la migration d'un microprocessus, la communication est alors faite entre deux nœuds, via un réseau. Le passage par le réseau a un très grand impact sur la façon d'échanger les données.

Dans cette situation, la même référence vers l'objet à échanger ne peut plus être envoyée. Une référence est l'équivalent d'une adresse mémoire pointant vers l'objet. Elle est donc fortement liée au nœud à l'intérieur duquel elle existe. Il est alors impossible de n'envoyer qu'une référence au nœud distant, ce dernier n'ayant pas accès à la mémoire du nœud qui effectue l'envoi. Une copie de l'objet est alors nécessaire.

Une transmission par le réseau se comporte de la même façon qu'une communication via un tube. On ne peut transférer qu'une suite d'octets qui sont lus et décodés par le récepteur. La copie est donc créée par l'encodage de l'objet original, sous une forme pouvant être transmise via un réseau *TCP/IP*, suivi du décodage pour recréer l'objet à l'intérieur du nœud de destination. Ces étapes peuvent heureusement être masquées au programmeur, à l'intérieur de l'implémentation des canaux de communication.

Avec l'ajout des étapes d'encodage et de décodage, il devient par contre impossible d'envoyer tous les types d'objets par un canal réseau. Seuls les objets sérialisables peuvent être envoyés, ce qui exclut tous les types d'objets utilisés pour les entrées et sorties, tels les fichiers, les connexions réseau, l'entrée standard et la sortie standard. Cette limitation provient du fait que les entrées et sorties sont fortement liées à l'instance d'un programme dont elles font partie.

La dernière contrainte se situe par rapport à la possible topologie des utilisateurs

d'un canal de communication. Étant donné qu'un canal de communication peut être utilisé par plusieurs microprocessus, et que seule une partie d'entre eux peut être migrée vers un autre nœud, les canaux doivent gérer le cas où la communication peut soit être locale, soit être distante. Les canaux de communication doivent donc être hybrides.

Le comportement de base, où le premier microprocessus effectuant une requête sur le canal est le premier servi, doit être respecté et ce, peu importe si le microprocessus est sur un nœud distant ou non. Lorsqu'un microprocessus est bloqué sur une requête, le nœud doit alors informer tous les autres nœuds qui utilisent le canal qu'une requête est en attente. Le microprocessus est aussi placé dans la file d'attente du canal en attente d'une réponse.

Cette réponse peut alors provenir d'un autre microprocessus sur le même nœud ou d'un microprocessus exécuté sur un autre nœud. Le canal doit donc pouvoir ajouter à sa file d'attente des requêtes locales et des requêtes distantes, pour respecter l'ordre auquel il doit répondre à ces requêtes. Chaque nœud doit avoir sa propre file d'attente pour un canal donné. Cette duplication des files d'attente évite qu'un canal de communication soit dépendant envers un nœud en particulier. Les nœuds doivent donc être en constante communication, pour que ces files d'attente soient mises à jour au cas où une requête en attente aurait été répondue par un microprocessus se trouvant sur un autre nœud.

2.3.4 Résumé

Pour conclure cette section, nous nous retrouvons donc avec des canaux de communications distribués. Ces canaux de communications ont les propriétés suivantes :

- Ils sont bidirectionnels. Un microprocessus qui envoie sur un canal peut éventuellement recevoir sur le même canal.
- Le mode de communication est synchrone. Lorsqu'un message ne peut être envoyé ou reçu, le microprocessus ayant fait la requête est bloqué.
- La communication se fait sous forme d'échange de références ou de copies d'objets.
- La communication est atomique au niveau des objets. Un objet envoyé est donc assurément reçu en entier.
- Pour un canal donné, il peut y avoir plus d'un microprocessus en écriture et plus d'un microprocessus en lecture, qu'ils soient sur le même nœud ou non.
- Un système de file d'attente gère l'ordre d'arrivée des requêtes d'envoi et des requêtes de réception.
- Lorsque la communication a lieu via un réseau, le programmeur n'a pas à se soucier de l'encodage et du décodage de l'objet envoyé. Cette étape est à la charge du

canal.

La principale contrainte à laquelle le programmeur doit porter attention est la différence entre une communication locale à un nœud et une communication distante entre deux nœuds. Dans le premier cas, la transmission se fait à l'aide d'une référence et dans le second cas, à l'aide d'une copie de l'objet. Dans le cas de la transmission par copie, une contrainte s'ajoute : seuls les objets sérialisables peuvent être envoyés.

Un modèle de communication unique, soit par copie ou par référence, serait possible pour couvrir les échanges locaux à un nœud et à distance entre deux nœuds. Plusieurs facteurs sont par contre à considérer en lien avec la performance et la complexité d'un tel modèle. Des outils peuvent tout de même être offerts au programmeur pour qui le modèle de communication mixte ne convient pas.

2.4 Dépendances

Étant donné que l'idée de migration de microprocessus permet de créer des nœuds très légers, ces derniers n'ont possiblement pas tout le code nécessaire au fonctionnement des microprocessus qui leur sont envoyés. Le problème est le même lors de l'envoi d'un objet à un microprocessus exécuté sur un nœud distant, via un canal de communication. Le nœud n'a pas nécessairement à sa disposition la classe dont il a besoin pour recharger l'objet et l'utiliser. Un mécanisme de récupération des dépendances doit donc être mis en place pour qu'un nœud, qui a besoin de recharger un objet dont il ne connaît pas la structure, puisse demander cette dernière au nœud qui a envoyé l'objet.

2.4.1 Méthodes de récupération des dépendances

Un mécanisme de récupération des dépendances risque d'être appelé lors de tout échange d'objet entre deux nœuds, lorsque celui qui le reçoit est incapable de le charger. Deux méthodes sont possibles pour répondre à ce besoin et chacune d'entre elles a ses avantages et ses inconvénients :

1. La première méthode consiste à effectuer une analyse récursive complète du code associé au microprocessus à envoyer. Cette analyse permet de retrouver la liste des bibliothèques dont le programme dépend, en incluant aussi les bibliothèques dont les dépendances du programme dépendent.
2. La seconde méthode consiste à effectuer une récupération tardive des dépendances.

Lorsqu'un nœud reçoit un objet à recharger, s'il n'a pas à sa disposition le code nécessaire à ce rechargement, il le demande au nœud ayant envoyé l'objet.

Analyse récursive des dépendances

L'analyse récursive permet de construire une liste complète des dépendances d'un programme. Dans ces dépendances, on retrouve tous les fichiers sources du programme, les bibliothèques fournies par le programmeur, mais aussi les bibliothèques faisant partie du langage de programmation. Lors de la migration d'un microprocessus, cette liste peut alors être envoyée au nœud récepteur, en même temps que le microprocessus.

Le nœud récepteur peut alors comparer la liste de bibliothèques reçue aux bibliothèques qu'il a déjà a sa disposition. Si certaines sont manquantes, le nœud peut les demander immédiatement au nœud envoyeur. Avec cette méthode, le nœud receveur ne dépend donc pas du nœud envoyeur, car toutes les dépendances du code auront été copiées en une seule opération.

Cependant, une analyse du code pour retrouver les dépendances ne tient pas compte du code qui est utilisé lors de l'exécution d'un microprocessus donné. En effet, un programme peut contenir le code de plusieurs microprocessus et chacun d'entre eux peut avoir ses propres dépendances. Les dépendances de tous ces microprocessus sont alors copiées même s'il n'y a qu'un seul d'entre eux qui est effectivement migré. Cette étape de copie des dépendances pourrait alors être allongée inutilement, si le microprocessus migré n'a que très peu de dépendances par rapport aux autres présents dans le programme.

Cette méthode élimine par contre le besoin d'un mécanisme de récupération des dépendances lors du chargement des objets reçus via des canaux de communication. Ceci est une conséquence de la communication à distance sur un canal, qui implique qu'une migration de microprocessus a eu lieu précédemment. Toutes les dépendances possibles, pour que le microprocessus qui reçoit un objet puisse le recharger, ont nécessairement été récupérées lors de la migration qui a précédé l'échange de l'objet.

Récupération tardive des dépendances

L'autre méthode possible pour récupérer les dépendances d'un microprocessus est une récupération tardive. Lorsqu'un nœud reçoit un objet et a besoin de le recharger,

il vérifie tout d'abord s'il a son code à sa disposition. S'il n'est pas disponible, il met en suspend le processus de rechargement de l'objet et demande alors au nœud qui l'a envoyé de lui envoyer son code. Une fois reçu, le chargement de l'objet peut être complété.

Cette méthode peut aussi être utilisée lorsque le code d'un microprocessus demande d'importer une classe ou une bibliothèque que le nœud ne connaît pas. Une demande est alors faite au nœud qui a envoyé le microprocessus, pour qu'il fournisse le code manquant. L'inconvénient de cette solution est principalement relié à la migration de microprocessus. Lors de la première migration, on peut demander immédiatement les dépendances nécessaires au rechargement des objets au nœud envoyeur. Comme le code pourrait avoir besoin de nouvelles dépendances, plus tard pendant son exécution, il faut que le nœud où il se trouvait initialement reste disponible.

Un autre problème survient lors des migrations subséquentes du même processus. Si le microprocessus est démarré sur un nœud A, qu'il est par la suite migré vers un nœud B et finalement vers un nœud C, lorsque son code a besoin de nouvelles dépendances, il ne peut les demander à B. Il doit les demander au nœud A. Comme le bout de code demandant les nouvelles dépendances n'a pas été exécuté sur B, ce dernier ne les a pas récupérées de A. Dans le cas des microprocessus, nous devons donc prendre en note l'identifiant unique du nœud d'origine et c'est toujours à celui-ci que doivent être demandées les nouvelles dépendances. Il doit donc toujours être disponible.

Le problème d'une récupération tardive peut être éliminé si le programmeur porte une attention particulière à la structure de son programme. S'il s'assure de charger toutes les bibliothèques et toutes les classes nécessaires au tout début de l'exécution d'un microprocessus, elles feront alors partie de son état d'exécution. Lors de la migration, les noms des dépendances chargées pourront alors être envoyés et le nœud récepteur les rechargera immédiatement avant de recharger le microprocessus. Les dépendances manquantes pourront alors être demandées au nœud envoyeur. Si cette façon de faire est imposée au programmeur, nous n'avons plus besoin de noter le nœud original du microprocessus. Toutes ses dépendances devraient alors le suivre lors de chacune de ses migrations.

2.4.2 Mise en mémoire cache des dépendances

Avec des nœuds légers qui n'exécutent que des microprocessus provenant d'autres nœuds, il est préférable de mettre en place une mémoire cache pour le système de récupération des dépendances. Au fil du temps, si une bibliothèque est demandée plusieurs

fois par un nœud, il peut alors utiliser une copie qu'il a gardée localement. Il faut toutefois porter attention aux mises à jour de code. Lorsqu'un nœud reçoit un objet ou un microprocessus pour lequel il a déjà une copie d'une dépendance en mémoire cache, il doit savoir si l'objet ou le microprocessus reçu a besoin d'une copie à jour de celle-ci.

La vérification de la nécessité de mettre à jour une dépendance gardée en mémoire cache peut se faire à l'aide d'une information supplémentaire, envoyée par le nœud qui envoie un objet. La date et l'heure de la dernière mise à jour, un numéro de version ou bien une somme de contrôle de la dépendance peuvent répondre à ce besoin. Le nœud récepteur peut alors comparer cette information avec celle de sa copie en mémoire cache et prendre une décision.

2.5 Variables globales et objets partagés

Lors de l'exécution d'un programme, il est possible que plusieurs microprocessus viennent qu'à partager l'utilisation de certains objets. Ce partage d'objets est causé par les échanges qui se font par référence, lorsque les canaux de communication sont utilisés par des microprocessus exécutés sur un même nœud. Rappelons-nous que ce comportement a été choisi pour être cohérent avec le mode de passage des paramètres d'appel de fonction, qui se fait par référence.

Un problème survient lors de la migration de microprocessus, par rapport aux objets partagés. Ce problème est le même qu'avec les échanges, via un canal de communication, entre deux nœuds. Une copie du microprocessus et de tout son environnement mémoire est créée sur le nœud où ils sont rechargés, compte tenu de leur passage via le réseau. Si un objet était partagé entre deux microprocessus, il ne l'est plus et le programmeur ne doit pas s'attendre à ce qu'une valeur mise à jour sur un nœud le soit sur un autre.

Par rapport au partage d'objets, cette section présente les différents cas pouvant être problématiques lors des migrations de microprocessus et des échanges via les canaux de communication. Une solution est aussi présentée pour donner le choix de la façon dont les objets doivent être échangés entre deux nœuds.

2.5.1 Environnement global

Tout d'abord, dans le but de simplifier les migrations et de bien définir ce que chacun des microprocessus a à accomplir, ils doivent être exécutés dans des environnements globaux séparés. Cette séparation est semblable à celle qui existe avec des processus qui ont été dupliqués. Chacun d'entre eux a alors sa propre mémoire. De cette façon, un environnement global n'est pas partagé et lors d'une migration, il peut facilement suivre le microprocessus auquel il est attaché.

Les variables globales sont toujours disponibles, mais seulement pour les fonctions et méthodes faisant partie d'un même microprocessus. Avec ce comportement, il faut donc utiliser les canaux de communication pour l'échange de données entre deux microprocessus. Des moyens doivent tout de même exister pour partager un canal de communication :

- La première méthode est de le passer en paramètre, lors de l'appel de la fonction utilisée comme point d'entrée d'un microprocessus.
- La seconde méthode est d'envoyer le canal à un microprocessus, via une communication sur un autre canal qu'il connaît déjà.

2.5.2 Partage et copie d'objets

Tenter de définir un comportement uniforme pour les échanges d'objets, à l'intérieur d'un même nœud et entre deux nœuds, peut sembler approprié d'un point de vue théorique. Cette uniformisation simplifierait beaucoup la tâche au programmeur. Par contre, d'un point de vue pratique, un unique comportement n'est pas idéal pour des raisons de performance. Reprenons nos deux cas, soient l'échange par référence à l'intérieur d'un même nœud et l'échange par copie entre deux nœuds.

Échanges locaux

D'un point de vue local, un échange par référence est la solution la plus rapide. Seul un pointeur tout petit en terme d'espace mémoire est échangé. Un échange par copie ajouterait une certaine lourdeur, car l'opération de copie prend un certain temps. De plus, ce ne sont pas tous les objets qui doivent ou peuvent être copiés.

Habituellement, seuls les objets sérialisables peuvent facilement l'être. Les objets qui contiennent des références vers des fichiers ouverts ou des connexions réseau ne le

sont pas, car ces références sont fortement liées au système sur lequel elles se trouvent et ne peuvent être copiées. Il est tout de même possible de sérialiser de tels objets, à l'aide de méthodes personnalisées. Ces méthodes sont souvent très fastidieuses à mettre en place et dépendent grandement du besoin du programmeur.

Finalement, une copie par défaut des objets immuables est inutile, car plus d'espace mémoire serait alors utilisé pour différentes copies d'une même valeur. Ces objets ne peuvent de toute façon pas être modifiés. Un microprocessus qui assigne une nouvelle valeur à une variable pointant vers un objet immuable ne change pas la valeur pour un autre microprocessus qui utilise le même objet.

Échanges à distance

D'un point de vue distant, un passage par référence implique que les objets sont toujours hébergés sur le nœud d'origine. Toutes les opérations, effectuées à l'aide de la référence d'un tel objet, demandent une communication réseau pour qu'elles soient effectuées à distance sur l'objet. C'est la meilleure solution pour des objets volumineux pour lesquels seule une petite partie serait à utiliser par un nœud distant. Par contre, pour de petits objets, ces opérations à distance ajoutent des délais.

Pour les objets immuables, tels les nombres et les chaines de caractères, ces délais surpassent le temps nécessaire à la copie. De plus, ces délais sont présents pour toute demande de lecture de la valeur de l'objet. Une copie temporaire doit de toute façon être créée pour que la valeur puisse être utilisée par le nœud qui la demande, et ce, chaque fois qu'il en a besoin.

Finalement, si un objet est accédé par une référence à distance et que le nœud qui l'héberge est arrêté, toutes les opérations sur cet objet échoueront. Si tous les objets utilisés par un microprocessus fonctionnent de cette façon, ce dernier devra nécessairement être arrêté, car les données dont il a besoin ne seront désormais plus disponibles. Ce cas n'est pas particulièrement appréciable lorsque l'on veut éviter une dépendance envers un nœud en particulier.

2.5.3 Objets distribués

Pour le cas de très gros objets, que ce soit sur un même nœud ou vers un nœud distant, une copie par défaut n'est pas préférable. Dans un environnement local, la

mémoire peut être limitée et une copie peut engendrer une mauvaise utilisation des ressources. Entre deux nœuds, la copie peut demander un temps de transfert très long par rapport aux possibles opérations à faire sur l'objet. Dans ce cas, seule une référence vers l'objet devrait être envoyée lors d'un échange via un canal de communication ou lors d'une migration de microprocessus.

Or, notre modèle d'échange d'objets dicte que cet échange a lieu par copie entre deux nœuds. Cette limite provient de l'impossibilité d'échanger une référence entre deux nœuds et de la transmission via un réseau qui impose la copie. Ce problème peut par contre être résolu en ajoutant une couche supplémentaire, permettant de transformer des objets en objets distribués.

Si le programmeur a besoin qu'un objet soit toujours échangé par référence, il peut alors utiliser une fonction qui permet de marquer un tel objet. Un objet référence est retourné et c'est ce dernier qui est utilisé par le programmeur pour accéder à son objet original. L'objet référence doit connaître en tout temps où se trouve l'objet original, pour y rediriger les opérations demandées. Si l'objet original se trouve sur le même nœud, les opérations sont effectuées directement sur celui-ci. S'il se trouve sur un autre nœud, toute opération effectuée sur l'objet référence engendre une communication réseau vers le nœud qui héberge l'objet original, pour que l'opération puisse être effectuée sur celui-ci.

L'objet référence peut alors être envoyé de façon locale ou distante sans que l'on ait à se soucier que l'échange soit fait par référence ou par copie. Il en va de même lors de la migration d'un microprocessus qui utilise un tel objet référence. Peu importe le nombre de copies, chacune d'entre elles doit se comporter de la même façon. Elles doivent toutes rediriger les opérations effectuées sur elles vers l'objet original.

Du côté de la gestion de la mémoire, un nœud est capable de savoir si un objet qu'il héberge n'est plus utilisé par le code qu'il exécute. Un ramasse-miettes s'occupe alors de libérer la mémoire utilisée par les objets devenus orphelins. Avec l'introduction des objets distribués, en plus des références locales, le nœud doit connaître toutes les références distantes vers ces objets.

Les objets distribués sont la plupart du temps implémentés sous forme de bibliothèques. L'interpréteur n'est donc pas conscient des références distantes qu'un système d'objets distribués peut avoir sur certains objets. Un ramasse-miettes distribué doit donc être mis en place pour libérer les objets distribués lorsqu'il n'y a plus de référence distante vers ceux-ci. Les nœuds doivent être en constante communication pour que les références vers un objet distribué soient connues par le nœud qui l'héberge.

Tout comme pour les microprocessus, le but ultime est de pouvoir éliminer la dépendance envers un nœud en particulier. Un objet distribué doit donc pouvoir être déplacé vers un autre nœud. Lors du déplacement, toutes les copies de l'objet référence, associées à l'objet qui est déplacé, doivent être mises à jour pour que les requêtes puissent être envoyées vers le nouveau nœud où il réside. En plus de la migration de microprocessus, nous nous retrouvons aussi avec une migration d'objets distribués. Nous avons donc tous les moyens nécessaires pour avoir un code complètement mobile.

2.5.4 Ramasse-miettes distribué

Avec un système d'objets distribués, il est nécessaire de mettre en place un ramasse-miettes distribué [32]. En effet, le ramasse-miettes de l'interpréteur où est hébergé un objet distribué n'est pas capable de savoir s'il existe des références distantes vers cet objet. La gestion des objets distribués et la mise en place du ramasse-miettes distribué doivent donc être effectuées au niveau du nœud. Le système de nettoyage des objets est simple et est divisé en deux rôles :

1. Un nœud qui héberge un objet distribué doit s'assurer de son nettoyage lorsqu'il n'est plus utilisé.
2. Un nœud qui accède à distance à un objet distribué doit informer le nœud hébergeur de son utilisation.

Rôle du nœud hébergeur

Avec un système de ramasse-miettes distribué, un nœud qui héberge un objet distribué doit s'assurer qu'il est toujours utilisé par les nœuds distants. Lorsqu'un objet distribué n'est plus utilisé par aucun nœud, il doit le supprimer.

Pour ce, le nœud hébergeur maintient une référence vers l'objet distribué dans une liste d'objets distribués. Cette référence permet d'éviter la suppression de l'objet, par le ramasse-miettes de l'interpréteur, lorsqu'il n'y a plus aucun microprocessus local au nœud y faisant référence. La liste permet aussi au nœud de retrouver l'objet distribué lorsqu'il reçoit des requêtes, en provenance des autres nœuds, pour celui-ci.

Une liste des nœuds utilisateurs, ainsi que la date et l'heure du dernier accès à l'objet, doit aussi être maintenue pour chacun des objets distribués. Un processus exécuté à intervalles réguliers peut alors retirer de cette liste les identifiants des nœuds qui n'ont pas accédé à un objet distribué pendant un laps de temps donné. Lorsque ce processus

de nettoyage détecte que la date et l'heure du dernier accès pour un nœud n'ont pas été mises à jour, suite à sa précédente exécution, il décide que ce nœud n'utilise plus l'objet.

Le processus de nettoyage supprime finalement la référence vers l'objet distribué lorsqu'il n'y a plus aucun nœud distant qui l'utilise. Le ramasse-miettes de l'interpréteur du nœud peut alors le supprimer, s'il n'est plus utilisé par aucun microprocessus local.

Rôle du nœud utilisateur

De son côté, un nœud qui utilise un objet distribué a comme rôle de s'assurer que le nœud qui héberge cet objet le garde en mémoire tant et aussi longtemps qu'il en a besoin. À intervalles réguliers, il doit informer les nœuds qui hébergent tous les objets distribués qu'il utilise, pour leur indiquer qu'il a toujours besoin de ceux-ci. Ces intervalles de notification doivent être plus courts que ceux du processus de nettoyage d'un nœud hébergeur. Un délai réduit assure que la date et l'heure du dernier accès sont mises à jour entre deux exécutions du processus de nettoyage.

Lorsqu'il n'y a plus de microprocessus qui utilise la référence à distance d'un objet distribué, le nœud doit tenir compte qu'il n'a plus besoin de l'objet. Il cesse alors d'envoyer les notifications régulières au nœud hébergeur. Cet ensemble des tâches à accomplir, par un nœud hébergeur et par un nœud utilisateur d'objets distribués, forme ainsi le ramasse-miettes distribué.

2.5.5 Résumé

Dans cette section, nous avons vu les différentes façons de transférer les objets via les canaux de communication et lors des migrations de microprocessus. Ce transfert peut se faire, soit par référence, soit par copie. À l'intérieur d'un même nœud, un transfert par référence est privilégié pour réduire l'utilisation de la mémoire et pour la performance. Entre deux nœuds, étant donné que le transfert doit passer par un réseau, une copie est obligatoire. Un échange par référence est tout de même possible, à l'aide d'objets distribués. Un tel objet n'existe que sur un seul nœud et un objet référence, qui lui est associé, est utilisé pour y accéder. Un ramasse-miettes distribué est nécessaire pour le nettoyage de ces objets. Au même titre que les microprocessus, les objets distribués peuvent être migrés d'un nœud à un autre.

2.6 Entrées/Sorties

Avec plusieurs microprocessus exécutés sur différents nœuds, qui ne se retrouvent pas forcément sur un même ordinateur, un choix doit être fait par rapport à la gestion des entrées et sorties, qui sont fortement liées à l'ordinateur sur lequel elles sont utilisées. Ces entrées/sorties comprennent principalement le clavier, la souris, l'affichage, les fichiers et les connexions réseau. Il n'y a pas de méthode miracle qui peut couvrir tout ce qui a été énuméré. Par exemple, l'affichage dans un terminal se comporte de façon différente de l'accès à une connexion réseau. Dans cette section, des méthodes de gestion sont donc présentées pour couvrir l'utilisation des entrées et sorties dans un contexte de migration de microprocessus.

Commençons tout d'abord par l'utilisation d'un terminal, qui couvre l'entrée et la sortie standard. Chaque langage de programmation a ses propres fonctions pour lire et écrire sur le terminal et les programmeurs sont habitués à ces dernières. Avec un programme divisé en une multitude de microprocessus, le programmeur peut vouloir que chacun d'entre eux affiche des messages de statut. Lorsqu'un microprocessus est migré sur un autre nœud, se trouvant sur un autre ordinateur, les messages seront donc évidemment affichés dans le terminal de ce nouveau nœud.

Le programmeur peut tout de même vouloir que les messages continuent d'être affichés dans le terminal du nœud où le microprocessus se trouvait initialement. Il serait donc utile d'avoir une méthode permettant de décider où la sortie standard doit être envoyée pour chacun des microprocessus. L'idéal est de ne pas présenter de nouvelles fonctions au programmeur. Il doit pouvoir continuer de faire appel aux fonctions standards du langage de programmation qu'il utilise.

Heureusement, pour répondre à ce besoin, la plupart des langages de programmation ont des méthodes pour intercepter l'entrée standard et la sortie standard. Le système utilisé pour le comportement de base peut être remplacé par un autre système. Ce dernier est appelé dès que les fonctions standards du langage de programmation sont utilisées. Le nouveau système peut alors décider, par exemple, d'écrire dans un fichier tous les messages destinés à la sortie standard.

Une configuration peut donc être attachée à chacun des microprocessus. Lorsque les entrées et sorties standard sont appelées par un microprocessus, le système qui les intercepte est alors en mesure de lire cette configuration et de réagir selon celle-ci. Voici quelques exemples de ce que cette configuration peut contenir :

– Une étiquette identifiant le microprocessus, pour qu'elle soit préfixée à tous les

messages affichés par celui-ci.
- L'identifiant d'un autre nœud, pour que tous les messages à afficher lui soient envoyés. C'est alors ce nœud qui les affiche dans son terminal.
- Le nom d'un fichier, pour que tous les messages y soient écrits. Ce fichier ne doit pas nécessairement se trouver sur le même nœud.

Avec un tel système, nous pouvons donc offrir au programmeur la possibilité de définir le comportement de l'entrée et de la sortie standard pour chacun de ses microprocessus. La figure 2.2 démontre la redirection des entrées/sorties pour un seul microprocessus.

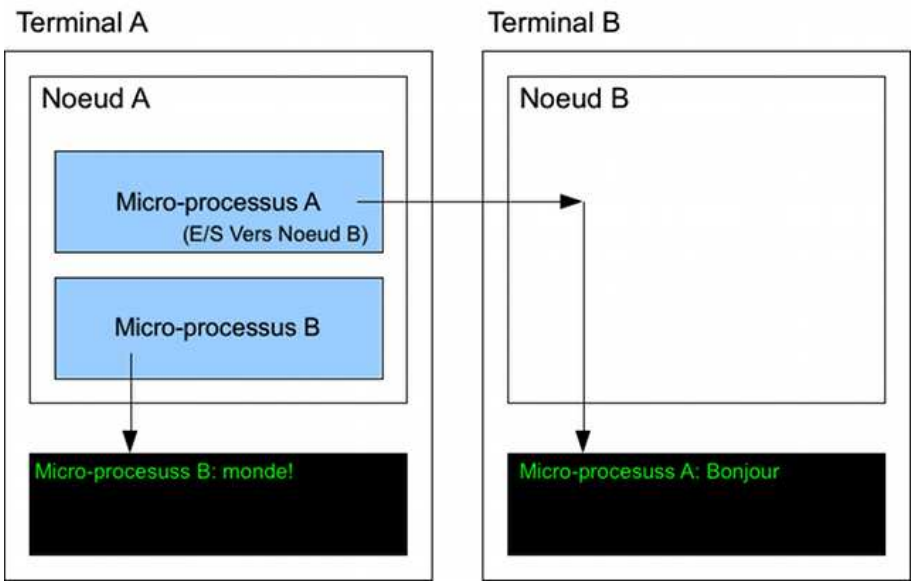

FIGURE 2.2 – Redirection de l'entrée et de la sortie standard vers un autre nœud.

Par rapport à la gestion des fichiers et des connexions réseau, la situation est beaucoup plus simple. Les mécanismes de base de la plupart des langages objets fonctionnent avec un objet retourné au programmeur. Il effectue alors des appels sur les méthodes de l'objet retourné pour lire et écrire sur le fichier ou sur la connexion réseau. Le mécanisme d'objets distribués introduit précédemment peut être utilisé pour gérer ce cas.

Un objet fichier, contenu dans un objet distribué, peut alors être librement envoyé à des microprocessus, exécutés sur des nœuds distants, et même être migré avec un microprocessus. Seul l'objet référence est alors envoyé aux nœuds distants. Les requêtes faites sur cet objet référence sont alors envoyées vers le nœud qui héberge le fichier. Une seule restriction s'applique par rapport à ce qui a été présenté pour le principe d'objets distribués : les objets d'entrées et de sorties ne peuvent être migrés vers un autre nœud, car ils sont fortement liés du système où ils se trouvent.

2.7 Gestion des erreurs

Pour gérer de façon simple les différents types d'erreurs qui peuvent survenir dans un programme, plusieurs méthodes existent. La méthode la plus ancienne est l'utilisation d'une valeur de retour numérique, lors de l'appel d'une fonction. Cette valeur peut indiquer, soit le succès de l'opération, soit qu'une erreur est survenue. Il faut donc la vérifier pour chaque appel de fonction et prendre une décision.

Avec les langages de programmation modernes, la gestion des exceptions remplace l'utilisation d'une valeur de retour. Cette méthode permet un contrôle plus précis des erreurs, à l'endroit où le programmeur décide de les gérer. De plus, cette méthode est souvent moins lourde à mettre en place et rend le code d'un programme plus facile à lire et à comprendre. Dans un monde distribué, cette méthode est aussi la plus simple à utiliser.

2.7.1 Valeur de retour de l'appel d'une fonction

Avant l'introduction de la gestion des exceptions dans les langages de programmation, la valeur de retour de l'appel d'une fonction, souvent sous forme de nombre, était utilisée pour indiquer les erreurs. En fait, la plupart des interfaces de programmation des systèmes d'exploitation utilisent toujours ce mode de fonctionnement. Une certaine valeur de retour, souvent *0*, indique qu'aucune erreur n'est survenue. Une autre plage de valeurs est utilisée pour indiquer toutes les possibles causes d'erreurs. La plupart du temps, chacune des valeurs est associée à une constante nommée pour que ce soit plus simple d'utilisation pour le programmeur. Avec ce mode de retour des erreurs, la gestion ressemble à ceci :

```
#include <stdio.h>

#define SUCCES                  0
#define ERREUR_PRIX_NEGATIF     1
#define ERREUR_AUCUN_RABAIS     2

int main()
{
    int prix;
    int rabais;
    int prix_final;
```

```
    int code_erreur;

    do {
        printf("Entrez le prix: ");
        scanf("%d", &prix);
        printf("Entrez le rabais: ");
        scanf("%d", &rabais);

        code_erreur = calcul_prix_rabais(
            prix,
            rabais,
            &prix_final
        );

        switch (code_erreur) {
            case ERREUR_PRIX_NEGATIF:
                printf("Le prix final ne peut " \
                    "être négatif!\n");
                break;
            case ERREUR_AUCUN_RABAIS:
                printf("Le rabais ne peut " \
                    "être nul!\n");
                break;
            case SUCCES:
                printf(
                    "Le prix final est: %d$\n",
                    prix_final
                );
        }
    } while (code_erreur != SUCCES);
}

int calcul_prix_rabais(int prix,
    int rabais, int *prix_final)
{
    *prix_final = prix - rabais;

    if (*prix_final == prix) {
        return ERREUR_AUCUN_RABAIS;
    } else if (*prix_final < 0) {
```

```
            return ERREUR_PRIX_NEGATIF;
        } else {
            return SUCCES;
        }
    }
```

Cette méthode comporte plusieurs désavantages. La valeur de retour de la fonction ne peut plus être utilisée pour retourner le résultat de l'opération. Il faut que ce résultat soit retourné par un paramètre d'appel, passé sous forme de référence ou pointeur. De plus, le programmeur doit gérer l'erreur immédiatement après l'appel de la fonction qui utilise cette méthode, car elle ne remonte pas la pile d'appels. S'il faut gérer l'erreur ailleurs dans le code, le programmeur doit lui-même faire remonter l'erreur à chacune des étapes de l'exécution. Ce travail est fastidieux et ajoute beaucoup de lourdeur au code du programme.

Finalement, seul un simple code d'erreur peut être retourné. Si le programmeur a besoin de retourner plus d'informations sur l'erreur qui est survenue, il doit utiliser d'autres moyens. Dans ce cas, une variable globale est souvent utilisée, car elle est accessible dans tout le programme. Cependant, avec l'utilisation d'une telle variable, la gestion d'une erreur devient alors déconnectée du contexte où elle est survenue. De plus, une telle variable globale est souvent partagée par les différents systèmes qui peuvent émettre une cause d'erreur. Le programmeur doit alors immédiatement copier la valeur de cette variable avant qu'une autre erreur survienne et qu'elle soit remplacée.

2.7.2 Gestion des exceptions

La plupart des langages de programmation offrent la gestion des exceptions. Une exception est un objet lancé lorsqu'une erreur survient. Lorsqu'une exception est lancée, elle remonte la pile d'appels des fonctions du programme, tant et aussi longtemps qu'elle n'est pas interceptée. Une exception peut être interceptée à même le programme, là où le programmeur décide qu'il peut la gérer. Si elle n'est pas interceptée, le programme termine et l'exception est affichée pour indiquer la cause de l'arrêt.

Lorsque l'on veut gérer les exceptions pouvant être levées dans un programme, la partie du code à gérer doit être contenue à l'intérieur de blocs *essayer/attraper/finalement* (*try/catch/finally*) . Les instructions contenues dans la partie *essayer* sont exécutées. Si une exception est levée, le flux d'exécution sort du bloc *essayer* et est dirigé vers le bloc *attraper* correspondant au type de l'exception. Le bloc *finalement* est

toujours exécuté, qu'une exception soit levée ou non.

Le type de l'objet exception représente le type de l'erreur qui est survenue. Différents types d'exceptions peuvent donc être interceptées par différents blocs *attraper*. Les détails de cette erreur peuvent aussi être ajoutés à l'objet exception pour aider le programmeur à la gérer. Voici un exemple de la structure de gestion d'une exception avec le langage Python :

```
try:
    traitement_pouvant_lancer_des_erreurs()

    autre_traitement()
except TypeErreur1, e:
    print e
except TypeErreur2, e:
    relancer_traitement(e.objet_a_traiter)
finally:
    effectuer_nettoyage()
```

Dans l'exemple précédent, on tente tout d'abord d'appeler la fonction *traitement_pouvant_lancer_des_erreurs*. Si une exception survient à l'intérieur de cette fonction, une exception est levée. Cette exception peut être soit de type *TypeErreur1* ou de type *TypeErreur2*. Une exception de type *TypeErreur1* est interceptée par le bloc suivant la ligne « except TypeErreur1, e : » (*except* correspond à *attraper*) et est tout simplement affichée. Une exception de type *TypeErreur2* est interceptée par la ligne « except TypeErreur2, e : ». Dans ce cas, la fonction *relancer_traitement* est appelée et l'objet *objet_a_traiter*, lié à l'objet exception, est passé en paramètre à cet appel.

Lorsqu'une exception est levée, la fonction *autre_traitement* n'est jamais appelée. Ceci est causé par le flux d'exécution qui sort du bloc *try* pour être redirigé vers un des deux blocs *except*.

Finalement, le bloc *finally* contient du code qui doit absolument être exécuté, qu'une exception soit levée ou non. Un tel bloc contient souvent du code pour libérer des ressources acquises à l'intérieur du bloc *try*, par exemple, de la mémoire ou un fichier.

2.7.3 Relation Parent/Enfant

Dans un scénario concurrent, une relation parent/enfant doit pouvoir exister entre deux microprocessus. Si un microprocessus enfant désire retourner le résultat d'un calcul à un microprocessus parent, il doit être informé si ce dernier meurt. Cet enfant pourrait alors terminer immédiatement son exécution. Dans le cas contraire, il deviendrait un processus orphelin et resterait alors bloqué indéfiniment lors de la tentative d'envoi du résultat au parent. Il polluerait ainsi l'espace mémoire du nœud où il se trouve tant et aussi longtemps que ce nœud n'est pas arrêté. Le besoin est le même si un enfant meurt. Le parent doit en être informé pour qu'il puisse éventuellement démarrer un nouvel enfant, qui pourra alors recommencer le calcul que le défunt n'a pu compléter.

La principale utilité de la relation parent/enfant est de rediriger les exceptions. Ces exceptions peuvent être générées par un microprocessus ou même par un nœud. Lorsqu'une relation existe entre deux microprocessus et que l'un d'entre eux lève une exception qui n'est pas attrapée à même le microprocessus, elle est alors envoyée à l'autre microprocessus faisant partie de la relation, où elle est levée. Ce dernier doit l'attraper et la gérer, à défaut de quoi il est interrompu à son tour.

2.7.4 Exceptions dans un environnement concurrent et distribué

Avec notre cadre de travail où plusieurs microprocessus sont exécutés en concurrence sur un même nœud, donc sur une même instance d'un interpréteur, nous devons faire une gestion très pointue des exceptions. Une exception lancée par un microprocessus ne doit absolument pas affecter l'exécution des autres microprocessus se trouvant sur le même nœud lorsqu'il n'y a pas de relation entre eux. Cette exception doit donc être interceptée par le mécanisme de gestion des microprocessus avant qu'elle ne remonte au niveau où l'interpréteur serait arrêté. Seul le microprocessus qui a lancé l'exception doit être arrêté, s'il ne l'intercepte pas. Le comportement est donc à l'image de celui des processus, au niveau du système d'exploitation.

La relation parent/enfant expliquée précédemment entre en jeu avec la gestion des exceptions. Certains microprocessus doivent pouvoir être mis en relation au cas ou l'un d'entre eux doit être informé de l'arrêt de l'autre. Celui informé de cet arrêt est alors débloqué s'il était en attente d'une information qui devait être envoyée par le microprocessus arrêté et peut alors décider quoi faire.

Si les deux microprocessus sont exécutés sur des nœuds différents, suite à une migration de l'un d'entre eux, l'exception doit être échangée entre les nœuds pour qu'elle puisse être levée à l'intérieur du microprocessus qui doit la recevoir. Dans cette situation, les nœuds doivent aussi gérer le cas où l'un d'entre eux est arrêté, terminant automatiquement le microprocessus qu'il exécutait. Le nœud restant doit détecter la perte de lien encourue par cet arrêt et lever une exception à l'intérieur du microprocessus qu'il exécute et qui était en relation avec le microprocessus distant, maintenant terminé.

Causes possibles d'arrêt d'un microprocessus

Lorsqu'une relation existe entre deux microprocessus, plusieurs causes sont possibles pour expliquer l'arrêt de l'un d'entre eux. Différents types d'exception peuvent alors être levés. Voici une description des différentes causes d'arrêt d'un microprocessus :

- Retour de la fonction utilisée comme point d'entrée lorsque le traitement est terminé.
- Une exception est lancée et n'a pas été interceptée par le microprocessus.
- Le microprocessus a été arrêté manuellement par l'injection d'une exception.
- Le nœud à l'intérieur duquel le microprocessus s'exécutait a été arrêté.

La fin du traitement d'un microprocessus ne justifie pas l'envoi d'une exception à l'autre membre de la relation. Ce cas survient normalement lorsqu'un traitement est complètement terminé et qu'aucune erreur n'est survenue. Il n'y a donc pas lieu d'informer le parent de quoi que ce soit. Un parent qui continue d'attendre une information de cet enfant par un canal de communication doit alors être considéré comme une erreur de programmation qui doit être corrigée.

Si un problème doit être mentionné au microprocessus parent, une exception peut être lancée à l'intérieur du microprocessus enfant à terminer. Lorsqu'une exception est levée dans un microprocessus enfant, elle est alors envoyée au parent si elle n'a pas été interceptée à l'intérieur du microprocessus qui l'a levée.

La même chose survient si une exception est explicitement injectée à l'intérieur d'un microprocessus pour forcer son arrêt. L'injection peut être utilisée lorsque, par exemple, le programmeur a besoin d'arrêter manuellement un microprocessus à partir d'un autre. Le microprocessus a alors la chance de terminer l'opération en cours ou d'effectuer un nettoyage des ressources qu'il utilise avant de terminer son exécution.

Pour ce qui est du dernier cas, le rôle revient au nœud de détecter la perte de lien entre deux microprocessus exécutés sur deux nœuds différents. Il s'occupe alors d'injecter une exception dans le microprocessus qu'il exécute et qui doit être mis au courant de la disparition de l'autre membre de la relation.

Injection d'exceptions

Lorsqu'une exception doit être transmise à un des deux membres d'une relation parent/enfant, un système permettant d'injecter des exceptions à l'intérieur d'un microprocessus peut être utilisé. Lorsque le nœud intercepte une exception ayant causé l'arrêt d'un microprocessus enfant, il retrouve la référence vers le microprocessus parent et lui injecte une nouvelle exception indiquant la raison de l'arrêt de son enfant.

La nouvelle exception injectée dans le microprocessus parent doit contenir une référence vers l'exception qui a causé l'arrêt du microprocessus enfant, ainsi qu'une référence vers celui-ci. Avec cette information, le parent est alors en mesure de connaître lequel de ses enfants s'est terminé et quelle en est la cause. La figure 2.3 explique ce processus.

2.8 Conclusion

Ce chapitre a donc présenté la mise en relation de différents concepts permettant de créer un système distribué basé sur la migration de microprocessus, dans un langage à mémoire partagée. L'assemblage de ces concepts n'est pas sans problème et dans beaucoup de cas, plusieurs solutions sont possibles. Des méthodes et outils ont été proposés pour couvrir la majorité des solutions possibles et pour donner le choix lors de l'utilisation du système.

Les microprocessus ont tout d'abord été situés par rapport aux processus et fils d'exécution du système d'exploitation. Ce sont des processus légers exécutés à l'intérieur d'un processus complet. Étant basé sur les continuations, il est possible de sauvegarder l'état d'exécution d'un microprocessus pour le récupérer plus tard. C'est ce qui permet leur migration.

Pour être migré, les microprocessus ont besoin d'un contenant, le nœud. Un nœud est l'interpréteur d'un langage interprété auquel des outils de communication sont ajou-

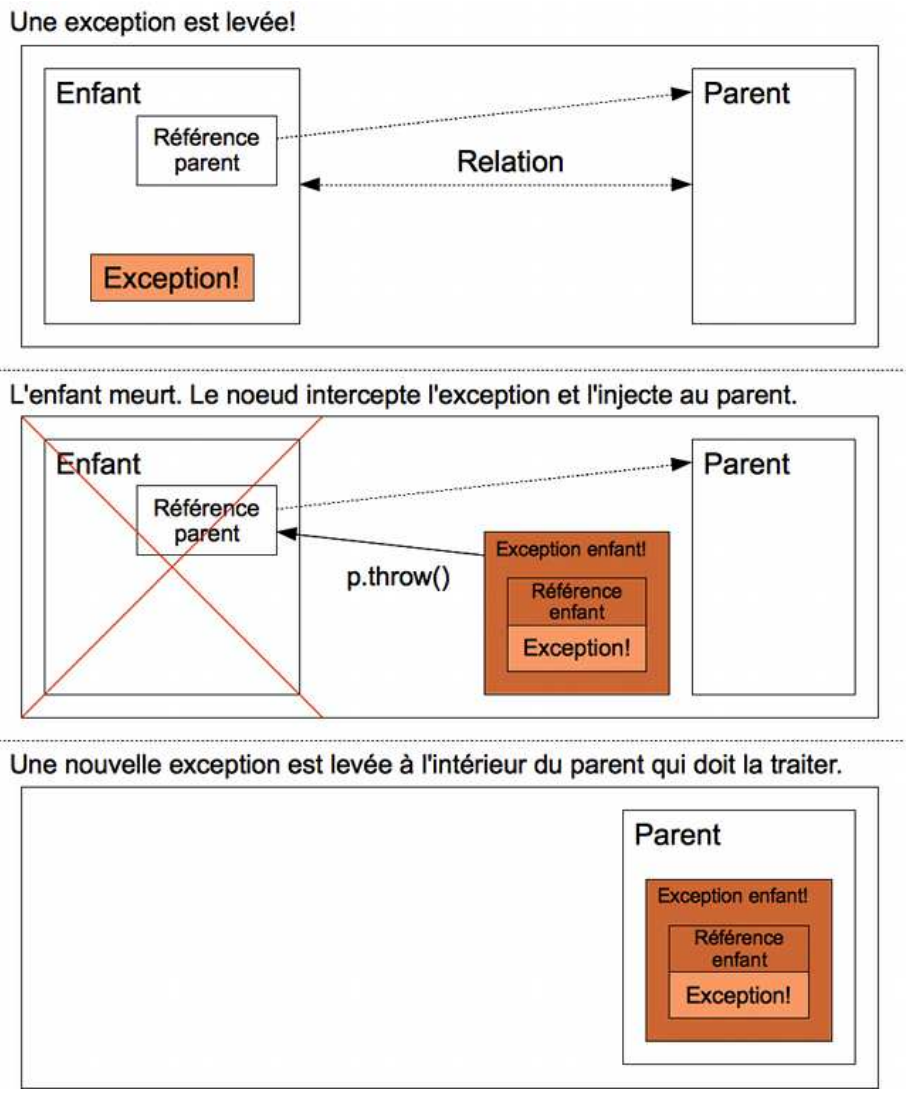

FIGURE 2.3 – Redirection d'une exception vers un microprocessus parent.

tés pour qu'il puisse communiquer avec d'autres nœuds. Un ensemble de nœuds forme un système distribué. L'utilisation d'un langage interprété permet de simplifier la migration de microprocessus. L'interpréteur a un contrôle complet sur l'exécution des microprocessus et il masque les différences des plates-formes matérielles et logicielles pour lesquelles il existe. Lorsque l'état d'exécution d'un microprocessus est sauvegardé, il peut donc être rechargé sur tout autre ordinateur pour lequel l'interpréteur est disponible.

Des canaux de communication permettent l'échange d'objets entre microprocessus. Un mode de communication synchrone a été choisi, car l'aide de l'ordonnanceur du nœud est nécessaire pour ce mode. Lorsqu'un microprocessus effectue une requête qui

ne peut être complétée immédiatement sur un canal, il doit être bloqué et retiré de la file d'attente de l'ordonnanceur. Si un mode de communication asynchrone est nécessaire, un nouveau microprocessus peut être démarré pour s'occuper de la communication bloquante. De cette façon, le microprocessus ayant amorcé la communication n'est pas bloqué. L'échange se fait par référence à l'intérieur d'un seul nœud et par copie entre deux nœuds.

Lorsqu'un nœud reçoit un ou plusieurs objets, suite à la migration d'un microprocessus ou suite à un échange via un canal de communication, il est possible qu'il n'ait pas le code nécessaire pour leur rechargement. Pour répondre à ce besoin, deux solutions sont possibles. La première est une analyse complète du code du programme pour envoyer la liste des dépendances en même temps que les objets. Cette façon de faire impose l'échange de code qui ne sera possiblement pas utilisé par les objets échangés. La deuxième façon de faire, qui a été retenue, est la récupération tardive des dépendances. Le nœud receveur demande seulement ce dont il a besoin pour recharger les objets reçus. Une mémoire cache permet aussi de limiter ces demandes.

Dans un contexte de migration de microprocessus, le partage d'objets est problématique. Ce partage, qui survient suite à l'échange via un canal de communication entre deux microprocessus locaux à un noeud, est difficile à éviter pour des raisons de performances. Lors d'un échange entre deux noeuds, suite à la migration d'un microprocessus ou à une communication, un objet partagé doit par contre être copié, car différents nœuds ne peuvent pas partager un même espace mémoire. Une solution unique n'est pas possible pour ce problème, car elle ne peut pas couvrir tous les besoins. Un système d'objets distribués, permettant d'accéder à des objets à distance, offre une certaine souplesse lorsque la copie n'est pas souhaitable.

Les entrées et sorties sont fortement liées au nœud à partir duquel elles sont utilisées et ne peuvent pas être migrées d'un nœud à un autre. Dans le cas, de l'entrée standard et de la sortie standard, les langages de programmation offrent un système permettant de les intercepter. Il est donc possible de mettre en place un tel système, qui permet de décider comment les gérer pour chacun des microprocessus. Dans le cas des fichiers et des connexions réseau, plus complexes à gérer, les objets distribués permettent d'y accéder à distance.

Finalement, le traitement d'un microprocessus peut être dépendant du traitement d'un autre microprocessus. Une gestion des erreurs doit permettre à deux microprocessus en relation d'être informé d'un éventuel problème avec un des deux membres. Lorsqu'une exception est levée par un microprocessus et qu'elle n'est pas interceptée, elle doit être envoyée à l'autre membre de la relation pour qu'il puisse être informé du

problème.

Le prochain chapitre repasse chacun de ces concepts et explique leur implémentation à l'aide du langage Python. Cette implémentation couvre tous les détails techniques nécessaires à la mise en place d'un module Python. Des exemples d'utilisation de ce module sont par la suite présentés.

Chapitre 3

Implémentation

Dans ce chapitre, nous voyons les détails de l'implémentation des concepts introduits précédemment, permettant la mise en place d'applications distribuées basées sur la migration de microprocessus. Notre objectif est d'étendre un langage existant, très connu de la communauté, qui se rapproche assez de la solution recherchée en terme de structure. Nous voulons être en mesure de prendre des programmes existants et de les transformer en programmes distribués avec peu de modifications dans leurs codes sources.

À cette fin, les fonctionnalités de Stackless Python ont donc été retenues. Cette version modifiée de l'interpréteur du langage Python offre un système de microprocessus nommés *tasklets*, ainsi que des canaux de communication permettant aux tasklets d'échanger des données. De plus, nous avons la possibilité de sérialiser l'état d'exécution de chacun de ces tasklets, ce qui fait une bonne base pour une migration de ceux-ci vers une autre instance de l'interpréteur.

Par ailleurs, l'implémentation PyPy du langage Python a été retenue, car elle supporte les fonctionnalités de Stackless Python, sous la forme d'un module nommé *stackless*. L'avantage principal de PyPy est que l'implémentation de ces fonctionnalités est faite en langage Python, contrairement à Stackless Python qui est programmé en langage C. De plus, cette implémentation est faite au niveau applicatif. Il n'y a donc aucune étape de compilation nécessaire pour tester les modifications et ajouts qui y sont apportés, ce qui sauve beaucoup de temps.

PyPy est donc beaucoup plus facile à comprendre et à étendre pour la mise en place de chacun de nos concepts. Le tout a été fait à l'intérieur d'un module nommé *dstackless*, pour *Distributed Stackless*. Il s'utilise de la même façon que le module *stackless* de

PyPy, avec très peu d'ajouts. Dans le but de bien comprendre la façon dont le module *dstackless* a été implémenté, ce chapitre présente en détail chacun de ces points :

- Une description du langage Python et de l'essentiel de sa syntaxe est tout d'abord faite dans le but de bien comprendre les exemples qui sont présentés.
- Une courte présentation de PyPy introduit ce que cette implémentation apporte par rapport à l'implémentation de base de Python.
- Les microprocessus sont construits à l'aide des tasklets de Stackless Python. Certaines informations ont été ajoutées à ces tasklets et une nouvelle méthode permet de les migrer facilement d'un nœud à un autre.
- Le nœud correspond à l'interpréteur de PyPy auquel certaines informations ont été ajoutées pour l'identifier dans un ensemble de nœuds. Un protocole de communication de bas niveau a dû être mis en place pour permettre aux différents nœuds de communiquer entre eux.
- Les canaux de communication s'utilisent de la même façon que ceux de Stackless Python. Le comportement de base est le même entre des microprocessus se trouvant sur un même nœud, mais les canaux sont transformés en canaux hybrides par l'ajout d'une couche de communication réseau permettant les échanges entre des microprocessus se trouvant sur des nœuds différents.
- Pour tout échange d'objet, incluant la migration de microprocessus, une copie de code peut s'avérer nécessaire d'un nœud à un autre. Le système d'importation de modules de Python permet d'intercepter les appels qui sont effectués a celui-ci, lors du rechargement des objets. On peut donc tirer profit de cette possibilité pour une récupération sur demande des dépendances.
- Le partage d'objets entre microprocessus introduit son lot de problèmes lorsque vient le temps de les envoyer à un autre nœud. Un système d'objets distribués aide à contourner certains de ces problèmes. Un outil intéressant offert par PyPy permet de simplifier la mise en place des objets distribués et de rendre leur utilisation la plus transparente possible pour le programmeur.
- La plupart des langages de programmation offrent des outils pour intercepter les entrées/sorties ; Python ne fait pas exception. Le système d'interception de Python est utilisé pour permettre la redirection des entrées/sorties pour chacun des microprocessus.
- Les fonctionnalités de Stackless Python n'offrent pas la possibilité de mettre en relation les microprocessus pour que les exceptions qui n'ont pas été interceptées dans un microprocessus soient redirigées vers un autre. Un ajout à PyPy a permis la mise en place de cette relation parent/enfant et de la redirection des exceptions.
- Pour terminer, l'interface de programmation du module *dstackless* est présentée.

3.1 Langage Python

Python est un langage interprété et à tout usage inventé en 1991 par Guido Van Rossum [28]. Il couvre les paradigmes de programmation objet, impératif et fonctionnel et se veut un langage simple d'apprentissage avec une syntaxe très légère mettant l'emphase sur la lisibilité. Comme la plupart des langages interprétés, il inclut un système de typage dynamique à l'exécution du programme, ainsi qu'une gestion automatique de l'allocation de la mémoire.

Python est utilisé la plupart du temps comme langage de script, mais il peut aussi être utilisé pour créer des applications riches et des applications web. Beaucoup de produits intègrent aussi le langage Python comme outil de création de scripts permettant à l'utilisateur d'ajouter des fonctionnalités. De par le fort intérêt envers ce langage, une très grande quantité de bibliothèques, permettant de simplifier le développement d'applications, est disponible.

3.1.1 Syntaxe

Dans le but de bien comprendre les exemples présentés dans ce chapitre, voici un résumé de l'essentiel de la syntaxe du langage Python [26].

Indentation

Au lieux des accolades ({ et }), que plusieurs langages utilisent, Python utilise l'indentation pour délimiter les blocs. Une augmentation de l'indentation signifie le début d'un bloc et une diminution de l'indentation signifie la fin d'un bloc.

Commentaire

Un commentaire commence par le symbole # et se termine à la fin d'une ligne.

Instruction sur plusieurs lignes

Pour qu'une instruction soit placée sur plusieurs lignes, chacune des lignes de celle-ci doit être terminée par le symbole \, à l'exception de la dernière ligne de l'instruction.

Variables et types

Le langage Python effectue un typage dynamique d'un programme. La vérification des types se fait donc à l'exécution, contrairement à un langage typé statiquement qui effectue cette validation à la compilation. Malgré cela, le typage qu'il effectue est fort. Une opération effectuée entre des types incompatibles lève une exception.

Une variable est définie lorsqu'une première valeur y est assignée. Elle prend alors le type de cette valeur. Le type d'une variable n'est pas fixe et peut changer si une valeur d'un type différent lui est assignée.

Les types suivants sont disponibles :

- **Chaîne de caractères** : Suite de caractères immuable (ne pouvant être modifiée). Une chaîne de caractères est contenue à l'intérieur de guillemets doubles (") ou de guillemets simples (').
- **Nombre entier** : Nombre entier immuable. Un nombre entier n'a pas de valeur minimale ou maximale définie par le langage, ni par le système. Elle se limite à la mémoire disponible dans le système.
- **Nombre à virgule flottante** : Nombre à virgule flottante immuable. La valeur est limitée par l'architecture matérielle du système.
- **Liste** : Une suite ordonnée et mutable (pouvant être modifiée) d'objets. Elle est déclarée à l'aide des crochets (*[* et *]*).
- **N-uplet (*tuple* en anglais)** : Une suite ordonnée et immuable d'objets. Elle est déclarée à l'aide des parenthèses (*(* et *)*).
- **Ensemble** : Un ensemble mutable et non ordonné d'objets uniques. Il est déclaré à l'aide de la classe *set*.
- **Dictionnaire** : Un ensemble mutable et non ordonné d'objets, chacun associé à une clé unique. Il est déclaré à l'aide des accolades (*{* et *}*).
- **Booléen** : Une valeur de vérité. Les valeurs possibles sont *True* et *False*.
- **Nul** : Une valeur nulle. Un seul objet de ce type, associé au mot-clé *None*, existe dans une instance de l'interpréteur.

Voici un exemple de déclaration de chacun des types :

```
# Chaîne de caractères.
c = "Bonjour monde!"

# Nombre entier.
i = 3

# Nombre à virgule flottante.
f = 3.2

# Liste.
liste = ["pomme", "banane", "orange"]
# Une liste peut être modifiée.
liste[1] = "poire" # Remplace "banane" par "poire".

# N-uplet.
nuplet = ("pomme", "banane", "orange")
# Un n-uplet ne peut être modifié.
nuplet[1] = "poire" # Génère une erreur.

# Ensemble.
ensemble = set(["pomme", "banane", "orange"])
print "banane" in x # Affiche "True".

# Dictionnaire.
dictionnaire = {"pomme": "rouge", "banane": "jaune"}
print dictionnaire["pomme"] # Affiche "rouge".

# Booléen.
b = False

# Valeur nulle.
n = None
```

Déclarations

Pour structurer le code d'un programme, plusieurs déclarations (mots-clés) sont utilisées. Elles permettent de gérer les conditions, d'effectuer des boucles et de structurer le programme en différentes classes et fonctions. Les plus courantes sont les suivantes :

- **if** : L'expression **if** (si) sert à effectuer un branchement dans le code d'un programme, selon une condition. Si la condition est vraie, le bloc associé à **if** est exécuté. Les mots-clés **elif** (sinon si) et **else** (sinon) sont utilisés pour les branchements alternatifs lorsque la condition est fausse.
- **for** : Effectuer une boucle pour chacun des éléments d'un objet itérable (liste, tuple, ensemble, etc.).
- **while** : Effectuer une boucle tant et aussi longtemps qu'une condition est vraie.
- **def** : Déclarer une fonction.
- **class** : Déclarer une classe.
- **try/except/finally** : Gérer les exceptions.
- **raise** : Lever une exception.
- **pass** : Instruction qui ne fait aucune opération. Cette instruction doit être utilisée lorsque le corps d'un bloc doit rester vide. Un bloc peut représenter le corps d'une fonction, d'une classe, d'une déclaration **if**, etc.
- **print** : Afficher un message à l'écran. Plusieurs variables peuvent être affichées sur une même ligne en les séparant par des virgules.

Voici un exemple d'utilisation de chacune de ces déclarations :

```
# Déclaration d'une nouvelle classe
# d'exception, qui étend
# la classe BaseException.
class ErreurDivisionZero(BaseException):
    pass

# Déclaration d'une fonction.
def diviser(a, b):
    if b == 0:
        # Lever une exception.
        raise ErreurDivisionZero()
    elif a == 0:
        # Le calcul n'est pas nécessaire.
        # Retourne toujours 0.
        return 0
    else:
        return a / b

def ne_rien_faire():
    pass
```

```
i = 3
try:
    while i >= 0:
        print "3 divisé par", i, "vaut", diviser(3, i)
        i = i - 1
except ErreurDivisionZero:
    print "Vous ne pouvez pas diviser par 0..."
finally:
    print "J'ai fini de diviser!"

# Parcourir une liste.
liste = ["pomme", "banane", "poire"]
for fruit in liste:
    print "Je mange une", fruit

# N'affiche et ne retourne rien du tout.
ne_rien_faire()
```

Expressions

Pour effectuer des opérations arithmétiques sur des variables ou pour les comparer, différentes expressions sont disponibles. Voici les principales :

Expressions mathématiques :

- + : Addition de deux variables de types numériques (entier ou flottant) ou concaténer deux chaînes de caractères.
- **-** : Soustraction de deux variables de types numériques.
- $*$: Multiplication de deux variables de types numériques.
- / : Division de deux variables de types numériques.

Expressions logiques :

- == : Teste l'égalité de deux variables. Retourne *True* en cas d'égalité des valeurs, *False* sinon.
- **!=** : Teste l'inégalité de deux variables. Retourne *False* en cas d'égalité des valeurs, *True* sinon.
- **is** : Vérifie si deux variables pointent vers le même objet.
- **in** : Vérifie la présence de la valeur d'une variable à l'intérieur d'une collection (liste, tuple, etc.).

- $>$: Vérifie qu'une variable a une valeur plus grande qu'une autre.
- $>=$: Vérifie qu'une variable a une valeur plus grande ou égale à une autre.
- $<$: Vérifie qu'une variable a une valeur plus petite qu'une autre.
- $<=$: Vérifie qu'une variable a une valeur plus petite ou égale à une autre.
- **and** : *Et* logique. Retourne *True* si les deux expressions logiques testées valent *True*.
- **or** : *Ou* logique. Retourne *True* si au moins une des deux expressions logiques testées vaut *True*.
- **not** : Inversion logique. Retourne l'inverse de la valeur de vérité à laquelle elle est associée.

Programmation fonctionnelle :

- **lambda** : Création d'une *lambda expression*. Elle peut être considérée comme une fonction anonyme. Le corps d'une *lambda expression* ne peut contenir qu'une seule instruction.

Voici un exemple d'utilisation de chacune de ces expressions :

```
# Addition.
a = 3
b = a + 4 # "b" vaut 7.

# Concaténation
bonjour = "Bonjour "
monde = "monde!"
bonjour_monde = bonjour + monde
# bonjour_monde contient "Bonjour monde!"

# Soustraction.
c = 3
d = c - 2 # "d" vaut 1.

# Multiplication.
e = 3
f = e * 3 # "f" vaut 9.

# Division.
g = 3
h = g / 3 # "h" vaut 1.

if d == h:
```

```
    print "Vrai" # "Vrai" est affiché.

if d != h:
    print "Vrai" # "Vrai" n'est pas affiché

x = "Bonjour"
y = x
if x is y:
    print "Vrai" # "Vrai" est affiché.

z = ["banane", "poire", "pomme"]
if "poire" in z:
    print "Vrai" # "Vrai" est affiché.

if b > f:
    print "Vrai" # "Vrai" n'est pas affiché.

if b < f:
    print "Vrai" # "Vrai" est affiché.

if (x is y) and (b < f):
    print "Vrai" # "Vrai" est affiché.

if (x is y) or (b > f):
    print "Vrai" # "Vrai" est affiché.

if not (x is y):
    print "Vrai" # "Vrai" n'est pas affiché.

# Déclaration d'une lambda expression.
l = lambda x, y: x + y
print l(3, 3) # Affiche 6.

# Une lambda expression peut être passée
# en paramètre à une fonction.
def executeur(valeur1, valeur2, fonction)
   return fonction(valeur1, valeur2)

# Affiche le résultat de 1 + 2
print executeur(1, 2, lambda x, y: x + y)
```

```
# Crée un tuple contenant les éléments
# "pomme" et "poire"
tuple = executeur("pomme", "poire", lambda x, y: (x, y))
```

Module

Dans le langage Python, chacun des fichiers source d'un programme est considéré comme un module. Le nom du module correspond au nom du fichier, en omettant l'extension *.py*. Lors du démarrage d'un programme Python, le fichier source utilisé comme point d'entrée est chargé à l'intérieur de l'interpréteur en tant que module, avec une particularité. Son nom de module est *__main__*, au lieu du nom du fichier.

Lors du chargement d'un module, toute instruction se trouvant à l'extérieur d'une fonction ou d'une classe est directement exécutée. Il est très simple de structurer un fichier source pour qu'il puisse être utilisé à la fois comme bibliothèque, et à la fois en tant que point d'entrée d'un programme. Pour différencier ces cas, il suffit d'effectuer un test sur le nom du module, retourné par la variable *self.__module__* : s'il est *__main__*, c'est que le fichier a été appelé en tant que programme. Voici un exemple :

```
# addition.py
def fonction_addition(a, b):
    return a+b

# Point d'entrée lorsqu'utilisé comme programme.
if self.__module__ == '__main__':
    print "Accès au point d'entrée du programme..."
    print "2 + 2 =", fonction_addition(2, 2)
```

Dans cet exemple, lorsque le fichier *addition.py* est appelé directement, à l'aide de la commande *python addition.py*, la variable *self.__module__* contient la valeur *__main__*. La condition est alors vraie et l'intérieur du bloc *if* est exécuté. Le résultat suivant est affiché :

```
Accès au point d'entrée du programme...
2 + 2 = 2
```

Le même fichier source peut aussi être utilisé comme un module et être importé à l'intérieur d'un autre module Python. Pour ce, les mots-clés *import* et *from [module]*

import [nom] sont utilisés. Les deux exemples suivants montrent l'utilisation de ces mots-clés pour importer le module *addition.py* et appeler la fonction *fonction_addition*. Voici le premier exemple :

```
import addition

# Le nom du module doit être spécifié pour
# appeler "fonction_addition".
print "3 + 3 =", addition.fonction_addition(3, 3)
```

Voici maintenant le deuxième exemple :

```
from addition import fonction_addition

# "fonction_addition" peut être
# appelée directement.
print "3 + 3 =", fonction_addition(3, 3)
```

Dans ces exemples, étant donné que *addition.py* a été importé en tant que module, la condition *self.__module__ == '__main__'* est alors fausse et les deux messages du bloc **if** ne sont pas affichés. Les deux exemples affichent donc le même résultat :

```
3 + 3 = 6
```

Fermeture

Une fermeture est une fonction contenant des variables libres pour lesquelles des valeurs sont attribuées, avant l'appel à celle-ci. Une telle fonction peut être vue comme un gabarit de fonction. La création d'une fermeture se fait par un appel à une fonction, qui retourne une référence vers une fonction *lambda* ou une fonction Python, après avoir attribué des valeurs aux variables libres. Voici un exemple :

```
def additionneur(valeur_a):
    def addition(valeur_b):
        return valeur_a + valeur_b

    # Retourne une référence vers une instance
    # de la fonction "addition"
    # pour laquelle valeur a été attribuée
    # à la variable libre  "valeur_a".
    return addition
```

```
# Création d'un additionneur qui additionnera
# toujours la valeur 5 à la valeur passée
# en paramètre.
fonction_addition = additionneur(5)

print "Le total est:", fonction_addition(3)
print "Le total est:", fonction_addition(5)
```

L'exécution de cet exemple affiche le résultat suivant :

```
Le total est: 8
Le total est: 10
```

3.1.2 Implémentations

Comme pour plusieurs langages interprétés il existe beaucoup d'implémentations de Python qui couvrent toute une panoplie d'environnements d'exécution et d'améliorations par rapport à l'implémentation de référence nommée *CPython* [29]. Les implémentations les plus connues sont les suivantes :

- IronPython : Compilateur pour la machine virtuelle *.NET* de Microsoft [11]. Il est possible de faire appel à du code d'un autre langage supporté par l'environnement *.NET* à partir de code Python exécuté sur cette implémentation.
- JPython : Compilateur pour la machine virtuelle du langage Java [12]. Il est possible de faire appel à du code Java à partir de code Python exécuté sur cette implémentation.
- Unladen Swallow : Implémentation améliorant la vitesse d'exécution [45].
- Stackless Python : Ajoute les concepts de tasklet et de canal de communication [35]. Les tasklets sont implémentés à l'aide de continuations et sont sérialisables.
- PyPy : Implémentation de l'interpréteur Python programmée en langage Python [22]. Elle offre les fonctionnalités de Stackless Python et certains outils expérimentaux qui ne sont pas disponibles dans aucune autre implémentation, mais qui sont tout de même très intéressants.

3.2 PyPy

PyPy est une implémentation du langage Python, de son interpréteur et de sa bibliothèque standard en langage Python. Cet interpréteur peut donc être exécuté à même une instance d'un autre interpréteur Python. Bien entendu, une exécution de PyPy à même une autre instance d'un interpréteur Python est extrêmement lourde et rend cette implémentation très lente lorsqu'utilisée de cette façon.

L'intérêt de PyPy provient cependant de l'ensemble des différents projets qui le composent et par la simplicité de son implémentation en code Python relativement à *CPython*, programmé en langage C. Plusieurs de ces projets visent à améliorer sa performance pour qu'elle devienne égale ou supérieure à celle de *CPython*. L'expérimentation de nouveaux concepts pouvant améliorer le langage Python est aussi encouragé, malgré le fait qu'ils ne soient pas toujours acceptés dans le projet *CPython* ; ce dernier visant la stabilité de l'interpréteur et du langage.

3.2.1 Traduction de code

Le premier projet d'intérêt est un traducteur de code Python vers d'autres langages. Le traducteur de référence génère du code C pouvant être compilé en fichier binaire, exécutable et autonome. Notons qu'il existe aussi des modules de traduction vers du code objet Java et *.NET* qui peut être exécuté sur les machines virtuelles respectives de ces environnements.

Pour pouvoir effectuer une transformation de code Python vers un autre langage, une analyse statique du code est effectuée afin de retrouver les types appropriés du langage de destination. Cette analyse statique n'est pas possible avec le langage Python standard compte tenu de sa syntaxe légère et de sa nature très dynamique. Un des facteurs qui empêche cette analyse statique est la possibilité de changer le type d'une variable lors de l'exécution d'un programme.

Des restrictions sont donc apportées au langage Python. Seuls les programmes écrits dans un sous-ensemble du langage pouvant être typé de façon statique peuvent être traduits : ce sous-ensemble se nomme *RPython*. L'implémentation du langage Python en code Python utilise donc le sous-ensemble *RPython*. L'interpréteur peut alors être traduit en code C et être compilé pour générer un exécutable autonome qui est presque aussi rapide que l'implémentation *CPython*.

3.2.2 Compilation à la volée

Dans le but d'améliorer la performance de l'interpréteur PyPy, le deuxième projet d'importance est la mise en place d'un compilateur à la volée (*Just-In-Time Compiler* en anglais) [23]. L'interpréteur de base interprète le code objet Python à l'aide de code C. De son côté, un compilateur à la volée tente, lorsque possible, de traduire le code Python en code machine. Un cache de ce code est aussi géré pour les utilisations subséquentes.

L'appel à la couche programmée en C est donc éliminée avec un compilateur à la volée. Ce processus accélère considérablement l'exécution de code Python et permet donc à l'interpréteur PyPy d'être plus rapide que *CPython*. Ce mécanisme est très utilisé par les machines virtuelles *.NET* et Java. Malheureusement, ce projet est encore jeune et le compilateur à la volée n'est pas utilisable avec toutes les configurations possibles de PyPy.

3.2.3 Gestion de la mémoire

Un autre changement apporté à PyPy, par rapport à *CPython*, est la façon dont les objets sont gérés en mémoire. Au lieu de n'avoir qu'un seul modèle de gestion, PyPy met en place un mécanisme permettant de brancher et d'utiliser plusieurs modèles différents. Bien entendu, le modèle de gestion standard de *CPython* est implémenté. Pour tous les types du langage, ce modèle définit la façon dont les objets se comportent et la façon dont ils sont stockés en mémoire.

Un autre modèle de gestion des objets a été mis en place. Il s'agit d'un modèle par objet mandataire (*proxy* en anglais). Ce modèle ajoute une couche, qui utilise le modèle de gestion standard, mais qui permet de contrôler l'accès aux objets en mémoire. Cette nouvelle couche permet d'intercepter toutes les requêtes effectuées sur les objets et d'introduire de nouveaux comportements.

3.2.4 Stackless Python

Pour terminer, une implémentation des fonctionnalités de Stackless Python a été mise en place sous forme d'un module nommé *stackless* [25]. Cette implémentation est écrite en très peu de lignes de code Python et est très facile à comprendre. De plus, les fonctionnalités du module *stackless* peuvent facilement être étendues par la création

d'un nouveau module, qui hérite de celui-ci.

3.2.5 Résumé

Toutes les améliorations que PyPy apporte au langage Python et la simplicité avec laquelle il est possible de le faire évoluer font en sorte que cette implémentation nous semble être un choix judicieux pour la mise en place de nos concepts à même un langage de programmation.

Notre modèle distribué par migration de microprocessus a été implémenté à l'aide d'une extension au module *stackless* de PyPy. Le fonctionnement de base de ce module est réutilisé pour l'exécution des microprocessus, ainsi que la communication entre ceux-ci, dans un nœud. Les nouvelles fonctionnalités couvrent principalement la migration des microprocessus ainsi que la communication entre eux, lorsqu'ils sont exécutés sur plusieurs nœuds.

Finalement, pour mettre en place le fonctionnement des objets distribués, le modèle de gestion des objets par objet mandataire, spécifique à PyPy, a été exploité. La possibilité d'intercepter les opérations effectuées sur un objet mandataire nous permet de les rediriger vers un autre nœud. Celui-ci peut alors les exécuter sur l'objet réel, auquel l'objet mandataire est associé.

3.3 Microprocessus

Notre implémentation des microprocessus se base sur la classe *tasklet* présente dans le module *stackless* de PyPy. Une nouvelle classe, présente dans notre module *dstackless* est aussi nommée *tasklet* et hérite directement de celle du module *stackless*. Quelques fonctionnalités y sont ajoutées pour supporter la migration de ceux-ci, d'une instance de l'interpréteur à une autre. Cette section couvre les points suivants, permettant de comprendre de quelle façon nos microprocessus ont été implémentés :

- Pour débuter, une description des fonctionnalités des tasklets est présentée. Cette description permet de comprendre leur fonctionnement et pourquoi nous pouvons les utiliser pour implémenter nos microprocessus.
- Dans un système distribué, les microprocessus doivent être identifiés de façon unique. Cet identifiant est utilisé lorsque des opérations doivent être effectuées à distance sur ceux-ci.

– Pour effectuer la migration d'un microprocessus, une nouvelle méthode, nommée *move*, a été ajoutée à la classe *tasklet*. Toutes les étapes et les outils nécessaires à cette migration sont présentés.
– Lors de la migration d'un microprocessus, son code et son environnement mémoire doivent être sérialisés. Les contraintes reliées aux étapes de sérialisation et de désérialisation sont expliquées.

3.3.1 Tasklet

Avec Stackless Python, un *tasklet* est l'implémentation d'un microprocessus. C'est un petit processus léger qui est exécuté en concurrence collaborative avec d'autres tasklets, sur une seule instance de l'interpréteur. Les tasklets sont ordonnancés à tour de rôle et chacun d'entre eux garde la main tant et aussi longtemps qu'il en a besoin. Un appel à la fonction *stackless.schedule* permet de donner la main au prochain tasklet dans la file d'attente.

La création d'un objet *tasklet* demande une fonction et des paramètres qui seront utilisés par la fonction au démarrage du tasklet. Une fois créé, un tasklet est automatiquement ajouté à la fin de la file d'ordonnancement. Un appel à *stackless.run* est alors nécessaire pour démarrer l'ordonnanceur. Par ailleurs, un tasklet peut être créé à tout moment, à partir du code d'un des tasklets en cours d'exécution ; il est ajouté à la file d'ordonnancement et sera exécuté quand son tour viendra.

Lorsqu'un tasklet est bloqué sur un canal de communication, il est retiré de la file d'ordonnancement. Il est alors placé dans une file d'attente du canal de communication. Il ne sera donc pas exécuté tant et aussi longtemps que la communication ne pourra être complétée. C'est de cette façon que le mode de communication synchrone est mis en place. Le programmeur peut aussi interrompre un tasklet à l'aide de la méthode *objet_ tasklet.remove* et l'insérer à nouveau dans la file d'attente à l'aide de la méthode *objet_ tasklet.insert*. Un tasklet peut aussi être arrêté avec un appel à *objet_ tasklet.kill*. L'exception *TaskletExit* est alors levée à même le tasklet à arrêter. Ce dernier peut tout de même intercepter cette exception et ainsi empêcher son arrêt.

Pour notre environnement distribué, où des microprocessus peuvent éventuellement être migrés vers d'autres nœuds, les tasklets ont de très gros avantages. Pour la mise en place des ceux-ci, l'implémentation Stackless Python élimine l'utilisation de la pile du langage C pour utiliser à la place un système plus efficace de continuations. L'état d'exécution d'un tasklet est alors simplifié et l'interpréteur a maintenant la capacité de le sauvegarder sous une forme indépendante de la plate-forme matérielle et logicielle sur

laquelle il est exécuté. Une fois cet état sauvegardé, il peut être rechargé sur la même instance de l'interpréteur ou sur une autre instance, qui peut se trouver sur une plate forme matérielle et logicielle différente sans causer de problème.

3.3.2 Identifiant unique

Dans un environnement où seul un interpréteur est impliqué, les objets *tasklet* peuvent être référencés directement pour que l'on puisse effectuer des opérations sur ceux-ci. Cette référence est interne à l'interpréteur et ne peut être envoyée à un autre interpréteur. En présence d'un environnement distribué, où plusieurs interpréteurs communiquent entre eux, un moyen doit être mis en place pour identifier un microprocessus sur lequel une opération doit être effectuée (possiblement à distance). Un identifiant unique, pour l'ensemble des interpréteurs interconnectés, doit donc être associé à chacun des microprocessus.

L'identifiant unique du nœud sur lequel un microprocessus est exécuté lui est aussi associé dans le but de le retrouver, sans avoir à recourir à un serveur de noms. Un interpréteur qui doit effectuer une opération à distance sur un microprocessus utilise donc un objet référence, qui contient tous ces identifiants. Il peut donc contacter le nœud, où le microprocessus est exécuté, en envoyant l'opération à effectuer, les paramètres de cette opération ainsi que son identifiant unique pour que le nœud le retrouve.

Pour simplifier la tâche de création d'un identifiant unique et pour éviter l'utilisation d'un serveur de noms, permettant de générer des noms uniques, le système UUID a été utilisé. Celui-ci permet de générer un identifiant unique composé de l'adresse de la première carte réseau du système, de l'heure courante ainsi que d'une partie aléatoire. Le risque de collisions est donc très faible, voir nul. Un module du langage Python, nommé *uuid*, offre déjà cette fonctionnalité et peut être utilisé sans ajouter une nouvelle dépendance à notre module.

3.3.3 Migration

La mise en place du concept de migration a été réalisée par l'ajout d'une méthode *move* à la classe *tasklet*. Cet ajout concorde bien avec le modèle objet pour lequel une action à prendre sur un objet doit être gérée par ce dernier. Un microprocessus doit donc être en mesure de se déplacer lui-même. La méthode *move* reçoit un paramètre qui correspond à l'identifiant unique du nœud vers lequel le microprocessus doit être

migré.

Lors d'un appel à *move*, on indique donc au microprocessus de se déplacer vers le nœud identifié. Son état est alors sauvegardé et récupéré sur le nouveau nœud. Son exécution cesse naturellement sur le nœud où il se trouvait initialement. Cette façon de faire nous permet donc de masquer toute la complexité derrière l'opération de migration et nous présentons au programmeur qu'une seule méthode simple à utiliser.

Étapes de la migration d'un microprocessus

Plusieurs étapes sont nécessaires pour la migration d'un microprocessus. En voici une description :

1. La première étape de la migration est de vérifier si le nœud courant dispose d'une connexion active avec le nœud de destination. Cette connexion est nécessaire pour envoyer l'état du microprocessus. S'il n'y a aucune connexion entre les deux nœuds, une tentative est effectuée, pour en créer une. En cas de succès, le processus de migration peut se poursuivre et dans le cas contraire, il se termine immédiatement.
2. Une fois la connexion obtenue, à l'aide du module pickle de Python, une sérialisation est effectuée sur l'objet *tasklet* représentant le microprocessus. Cette étape sert à sauvegarder l'état d'exécution, une copie des variables locales de toutes les fonctions et méthodes faisant partie de cet état et une copie de l'environnement global du microprocessus. C'est le résultat de cette sauvegarde, sous forme de suite d'octets, qui est envoyé au nœud receveur.
3. La prochaine étape est l'envoi d'un message *envoi de processus*, avec l'état sérialisé, au nœud qui doit recevoir le microprocessus. Seul l'envoi du message est validé. Aucune vérification n'est faite pour savoir si le microprocessus a bien été reçu, car le protocole *TCP/IP*, utilisé pour la communication, assure l'intégrité de l'envoi.
4. Une fois l'envoi terminé avec succès, l'exécution de la copie originale du microprocessus est terminée par un appel à la méthode *kill* de l'objet *tasklet*.
5. Suite au succès de la migration, l'objet *tasklet* original, pour lequel l'exécution est maintenant terminée, est transformé en référence distante. L'identifiant unique du nœud sur lequel se trouve maintenant le microprocessus est attribué à cet objet. Son type reste le même, mais l'ajout de cette nouvelle information lui indique qu'il doit changer son comportement. Toute opération effectuée sur cet objet référence est alors envoyée au nouvel hébergeur du microprocessus et c'est ce

dernier qui s'occupe de diriger cette opération au nouvel objet *tasklet* représentant le microprocessus.

6. Pour terminer l'opération, le nœud d'origine envoie une notification de migration à tous les nœuds qui sont connectés à celui-ci, donc qui ont potentiellement des références vers le microprocessus migré. Cette notification contient l'identifiant unique du microprocessus ainsi que l'identifiant unique de son nouveau nœud. Les nœuds qui reçoivent cette notification peuvent alors mettre à jour leurs références vers le microprocessus pour que les opérations soient dirigées vers le nouveau nœud sur lequel il se trouve.

Pour savoir si l'opération de migration a été effectuée avec succès, la valeur de retour de la méthode *move* est utilisée. Une valeur *True* indique que l'opération s'est déroulée correctement et que le microprocessus a été migré. Une valeur *False* indique que la migration ne s'est pas terminée correctement. Dans ce cas, le microprocessus continue son exécution comme si la demande de migration n'avait jamais eu lieu. En cas d'échec de l'opération de migration, lever une exception permettrait d'indiquer indiquer la cause de l'échec au programmeur.

Transformation d'un objet *tasklet* en référence

Une des étapes de la migration est la transformation de l'objet *tasklet* du microprocessus en référence distante. Le tout est géré à même notre implémentation de la classe *tasklet*. Ce changement de comportement de l'objet *tasklet* est nécessaire, car d'autres microprocessus du nœud d'origine peuvent avoir des références vers celui-ci, même si son état d'exécution a été migré.

Ces références sont des variables du langage Python et comme elles constituent un concept de bas niveau, au niveau de l'interpréteur, elles ne peuvent être modifiées en lot pour les faire pointer vers un nouvel objet d'un autre type. De plus, toute opération effectuée à l'aide de ces références doit être traitée par l'objet de la même façon que si le microprocessus se trouvait encore sur le même nœud.

La figure 3.1 illustre ce principe. Une fois *tasklet_a* migré, le microprocessus s'exécute maintenant sur un nouveau nœud. L'objet *tasklet_a* original existe toujours et contient maintenant une référence vers le nouveau nœud pour y rediriger les opérations effectuées sur celui-ci.

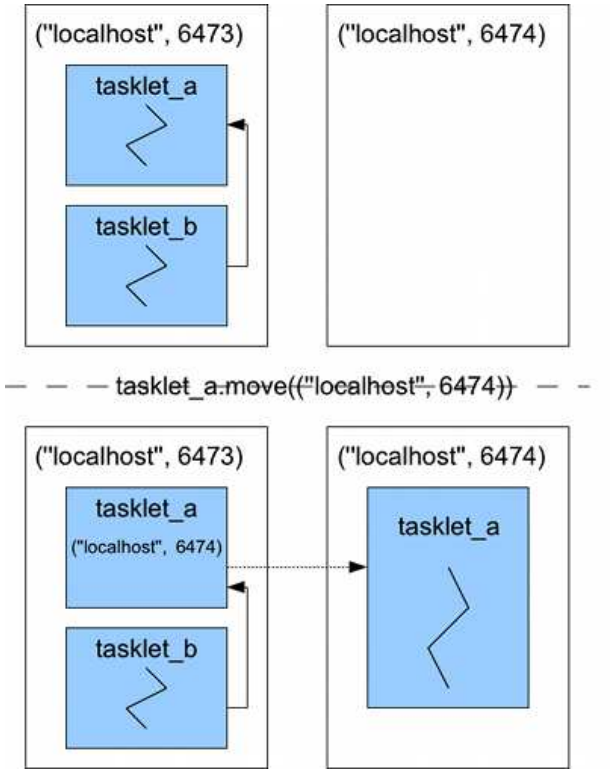

FIGURE 3.1 – Transformation d'un objet tasklet en objet référence.

Le cas contraire est tout aussi valide pour le nœud qui reçoit un microprocessus. Lorsqu'un objet *tasklet* est migré d'un nœud A vers un nœud B, qui a déjà une référence distante vers celui-ci, l'objet référence est réutilisé et est transformé en objet *tasklet* qui s'occupe alors de poursuivre l'exécution du microprocessus. Les autres microprocessus présents sur B utilisent désormais, sans aucune autre intervention, l'objet *tasklet* pour effectuer directement les opérations sur le microprocessus.

Cette récupération et transformation de l'objet référence est faite lors de la désérialisation. Lors de cette étape, si un objet référence est trouvé avec le même UUID que le microprocessus, il est réutilisé au lieu de créer un nouvel objet *tasklet*. L'état d'exécution lui est alors associé et l'identifiant unique du nœud distant qui exécutait le microprocessus est retiré. Cet objet *tasklet* est finalement inséré dans la file d'attente d'ordonnancement. Pour que ce système fonctionne, il faut s'assurer qu'un seul et unique objet *tasklet* existe sur chacun des nœuds, pour un microprocessus donné. Sur un nœud, cet objet est le microprocessus lui-même et sur les autres, une référence vers celui-ci.

3.3.4 Sérialisation

Lorsqu'une sérialisation est faite sur un objet *tasklet* deux comportements sont possibles :

1. Le premier cas survient lors d'une migration de microprocessus. L'appel à la méthode *move* ajoute un drapeau à l'objet qui permet alors au système de sérialisation de savoir qu'il doit sauvegarder l'état d'exécution.
2. Le deuxième cas survient lorsqu'un objet *tasklet* est envoyé via un canal de communication et que l'échange a lieu entre deux nœuds ou lorsqu'il est référé par un microprocessus migré. Dans ce cas, puisqu'un appel à *move* ne survient pas sur cet objet, le drapeau n'est alors pas levé : le système de sérialisation ne sauvegarde donc pas l'état d'exécution. À la place, l'identifiant du nœud actuel où est exécuté le microprocessus ainsi que le UUID de se dernier sont sauvegardés. Lors de la désérialisation, le nouvel objet *tasklet* créé correspond donc à une référence à distance vers le microprocessus qui n'a pas été déplacé.

Une attention particulière est à apporter lors de la désérialisation. Comme nous l'avons mentionné précédemment, un seul objet *tasklet* doit exister sur chacun des nœuds, pour un microprocessus donné. Lorsqu'un objet de ce type doit être désérialisé, nous devons tout d'abord vérifier s'il n'existe pas déjà un objet *tasklet* avec le même identifiant unique sur le nœud. Si cet objet existe, qu'il soit un objet référence ou le microprocessus lui-même, il est alors nécessairement référencé par un ou plusieurs microprocessus présents sur le nœud et doit être réutilisé. Autrement, un nouvel objet *tasklet* est créé.

Lorsqu'un objet est réutilisé, deux cas sont possibles par rapport à sa mise à jour :

1. Si l'objet à désérialiser est une référence à distance vers un microprocessus, aucune mise à jour n'a lieu de l'objet réutilisé.
2. Dans le cas où l'objet retrouvé est une référence à distance et que celui à désérialiser est un état d'exécution, une migration est donc en cours et cette référence est transformée en microprocessus, en lui attribuant l'état d'exécution et en retirant l'identifiant unique du nœud.

Le cas où l'objet retrouvé serait un microprocessus et que l'objet à désérialiser serait aussi un microprocessus ne peut survenir, car une migration est impossible à partir de, et vers un même nœud.

Pour la mise en place de nos comportements personnalisés, pour la sérialisation et la désérialisation, les surcharges qu'offrent le module pickle de Python ont été exploitées. Lorsque la méthode *__reduce__* est définie dans une classe, elle est appelée lors de la sérialisation et doit retourner une représentation de l'objet. Dans notre cas, elle retourne

l'état d'exécution ainsi que toutes les variables utilisées, lorsque la méthode *move* est appelée. Sinon, elle retourne l'UUID du microprocessus ainsi que l'identifiant unique du nœud qui l'héberge, pour qu'un objet *tasklet* référence soit créé lors de la désérialisation.

La représentation retournée par *__reduce__* peut indiquer une fonction, qui sera appelée afin de recréer l'objet. Cette fonction est alors appelée à la place du constructeur de la classe. Dans notre cas, cette fonction est spécifiée et lorsqu'elle est appelée, elle retourne, soit l'objet *tasklet* existant s'il est retrouvé dans un dictionnaire qui référence tous les objets *tasklet* présents sur le nœud à l'aide de leur UUID, soit un nouvel objet *tasklet*.

Une fois l'objet retourné, le processus de désérialisation fait un appel à la méthode *__setstate__*. Une surcharge de cette méthode nous permet de mettre à jour l'objet à l'aide de la représentation retournée par *__reduce__*. C'est à cette étape qu'un objet référence pourrait être transformé en microprocessus si une migration a eu lieu.

3.4 Nœud

L'implémentation du concept de nœud correspond à une instance de l'interpréteur PyPy, identifiée de façon unique dans un ensemble de nœuds. Il gère donc l'exécution des microprocessus ainsi que les échanges par les canaux de communication. Un nœud s'occupe aussi de gérer sa relation avec les autres nœuds. Cette relation inclut, par exemple, la synchronisation nécessaire lorsqu'un canal de communication est utilisé par plusieurs microprocessus exécutés sur différents nœuds, la migration d'un microprocessus et les échanges nécessaires pour l'accès à un objet distribué.

Pour les différents échanges entre nœuds, un protocole de communication comportant plusieurs messages a été mis en place. La connexion nécessaire pour que deux nœuds communiquent ensemble est gérée de façon automatique. Une connexion a lieu lors d'une demande de migration de processus ou lors de la réception d'une référence vers un objet partagé, comme un canal, ou hébergé sur un autre nœud, comme un objet distribué. Une référence peut être reçue lors d'un échange par un canal de communication ou à même l'environnement d'un microprocessus qui a été migré.

3.4.1 Identifiant unique de nœud

L'identification d'un nœud se fait à l'aide un n-uplet *([IP ou Nom d'hôte], Port)* (type *tuple* du langage Python), spécifié par l'utilisateur au démarrage de celui-ci. Cette utilisation d'un n-uplet permet d'assurer l'atomicité de l'identifiant. C'est ce n-uplet que le programmeur doit passer lors de l'appel d'une fonction ou d'une méthode, qui nécessite l'utilisation d'un identifiant unique de nœud. Une instance de l'interpréteur PyPy ne devient un nœud que lorsque le programmeur lui attribue son identifiant unique.

Tous les nœuds faisant partie d'un ensemble distribué doivent pouvoir se connecter aux autres nœuds à l'aide de leur identifiant. L'adresse *IP* doit donc être une adresse que tous les autres nœuds peuvent rejoindre. Aucune traduction d'adresse (*Network Address Translation* en anglais) n'est effectuée lorsqu'un nœud se trouve à l'intérieur d'un réseau privé. Lorsqu'il faut rejoindre un nœud faisant partie d'un tel réseau, une adresse publique doit être utilisée pour l'identifier. Un nom d'hôte, plus simple à retenir, peut aussi être utilisé. Il doit être associé à une adresse *IP* permettant de rejoindre le nœud. Une fois identifié, le nœud écoute sur le port faisant partie de l'identifiant : il est prêt à accepter des connexions en provenance d'autres nœuds.

3.4.2 Création d'un nœud

Une instance de l'interpréteur PyPy n'est pas considérée comme un nœud au moment de son démarrage. Cette instance est considérée comme un nœud suite à l'attribution d'un identifiant unique à celle-ci. La fonction *dstackless.identify* est utilisée pour l'identification. Une fois identifié, le nœud écoute automatiquement sur le port faisant partie de son identifiant unique. Les autres nœuds peuvent alors communiquer avec celui-ci.

Suite à une nouvelle connexion entre deux nœuds, chacun d'entre eux envoie à l'autre un message de type *IDENTIFY*, avec son identifiant unique, pour lui indiquer son identité. Pour les communications futures, le nœud qui reçoit un tel message associe alors la connexion réseau d'où il provient à l'identifiant unique reçu.

Voici un exemple de création d'un nœud simple, qui ne sert qu'à attendre que d'autres nœuds s'y connectent, pour lui envoyer des microprocessus ou des objets distribués :

```
from dstackless import identify, run

def main():
```

```
    # L'identifiant est passé en un seul
    # paramètre, sous forme d'un n-uplet.
    identify(("localhost", 6473))

    run() # Lance la boucle d'ordonnancement.

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

Un nœud plus complexe a cette même structure de base, mais peut aussi créer des canaux de communication, des microprocessus et des objets distribués. Les microprocessus peuvent alors être déplacés vers le nœud simple, si nécessaire. Suite à une migration de microprocessus, des objets peuvent maintenant être envoyés entre deux nœuds par un canal de communication partagé. Un objet distribué hébergé sur un des nœuds peut aussi être accédé à partir d'un autre nœud. Les différents nœuds impliqués s'occupent d'échanger les messages nécessaires, par le réseau, pour l'accomplissement de ces actions.

3.4.3 Protocole de communication

Le protocole *TCP/IP* est utilisé pour la création de liens entre les nœuds. Ce protocole ainsi que la volonté d'éviter la nécessité d'un serveur de noms force l'utilisation d'une adresse *IP* ou d'un nom d'hôte, et d'un port, pour identifier un nœud. Une fois une connexion *TCP/IP* établie, deux nœuds doivent avoir un langage commun pour échanger. Un protocole de plus haut niveau a été défini pour répondre à ce besoin.

La communication par une connexion *TCP/IP* se fait à l'aide d'un flot d'octets. Tous les messages envoyés et reçus doivent donc être encodés et décodés. Cette tâche devient très lourde lorsqu'il y a plusieurs messages de différentes tailles, avec différents types de données. De plus, chacune des lectures sur la connexion peut ne retourner qu'une partie des données. La suite est retrouvée lors des prochaines lectures.

Pour gérer tous ces problèmes, une couche de génération de protocoles réseau simple d'utilisation a été créée. Nous parlons de génération de protocoles réseau, car cette couche peut être rapidement configurée pour ajouter ou modifier les messages pouvant être envoyés. C'est ce qui a permis de faire évoluer facilement le protocole de communication au fur et à mesure que l'implémentation avançait.

Le générateur de protocoles est divisé en deux couches, permettant de bien séparer la communication de la gestion des messages faisant partie d'un protocole.

1. La couche de bas niveau s'occupe de la communication. Elle agit en tant qu'intermédiaire entre la couche définissant la structure du protocole de communication et le protocole *TCP/IP*.
2. La couche de haut niveau représente la structure d'un protocole. Les différents messages, ainsi que leurs paramètres, y sont définis. Cette couche varie d'un protocole à un autre.

Couche de communication

La couche de communication entretient un lien étroit avec le protocole *TCP/IP*. Son rôle principal est de masquer la complexité de son utilisation. Pour remplir ce rôle, cette couche a plusieurs tâches à accomplir. En voici une description :

- Elle offre une méthode pour écouter sur un port et ainsi agir en tant que serveur. Elle offre aussi une méthode pour se connecter sur une adresse *IP* et un port, et ainsi agir en tant que client.
- Une fois une connexion établie, pour assurer la synchronisation, une poignée de main est effectuée pour que les deux parties voulant communiquer vérifient l'utilisation de la même version du protocole. Une fois cette étape complétée, l'échange peut commencer.
- Un échange se fait sous la forme d'un message. Un message contient un nom, ainsi que plusieurs paramètres nommés, pouvant être de types différents.
- Pour envoyer un message sur une connexion *TCP/IP*, il doit être encodé sous la forme d'une suite d'octets. Cette couche s'occupe de cet encodage, ainsi que du décodage lorsque le message est reçu.
- Des vérifications sont faites à partir de la structure du protocole pour savoir de quelle façon encoder ou décoder un message. Cette vérification permet de rapidement détecter les erreurs de transmission.
- La lecture sur la connexion *TCP/IP* n'est pas automatique et se fait de façon non bloquante. À intervalle régulier, on doit indiquer à la couche de communication qu'il est temps de vérifier s'il faut recevoir des données. L'utilisateur du générateur de protocole doit donc appeler une méthode nommée *check_for_event*, à l'intérieur d'une boucle. C'est lors de cet appel que des messages peuvent être reçus.
- Lorsqu'un message a été décodé, un appel est fait à la couche de structure du protocole pour lever un événement. La méthode associée au message qui vient d'être décodé est appelée, avec les paramètres qui ont été décodés.

L'encodage sur le réseau suit des règles, qui permettent de simplifier le décodage. Le décodeur est toujours conscient de la quantité de données qu'il doit lire avant de considérer les différentes parties d'un message reçues et il peut rapidement détecter les erreurs. Voici comment un message est encodé :

- La structure d'un message ressemble à ceci (sans les parenthèses) : ((11)(NOM_MESSAGE)(7)(PARAM_1)(13)(BONJOUR_MONDE)(NUL)(7)(PARAM_2)(42)(NUL))
- Pour savoir comment encoder et décoder chacun des paramètres d'un message, une vérification est effectuée dans la structure du protocole pour connaître leurs types.
- Un nombre entier de 32 bits préfixe le nom du message, le nom de chacun des paramètres ainsi que leurs valeurs, à l'exception de la valeur des paramètres de type *entier* (aussi 32 bits). Ce nombre correspond au nombre d'octets qu'il y a à lire avant de conclure qu'il est possible de décoder une de ces parties d'un message. Pour les paramètres de type *entier*, la taille à lire est toujours de quatre octets.
- À la fin de chacun des paramètres d'un message, un caractère *nul* (valeur entière 0) est ajouté pour valider qu'ils aient bien été reçus.
- Pour les paramètres de type *tableau d'octets*, un caractère *nul* est ajouté à intervalle régulier. Lors de l'envoi d'une très grosse valeur, cet ajout permet de détecter les erreurs en cours de route et de terminer la connexion si une désynchronisation est détectée entre l'encodage et le décodage.

Pour l'encodage et le décodage des paramètres, la couche de communication offre plusieurs types. Ils sont tout d'abord utilisés lors de la définition des paramètres de chacun des messages d'un protocole. Du côté de la couche de communication, ils sont utilisés pour déterminer les méthodes d'encodage et de décodage à utiliser.

Une fonction spécifique à chacun des types est utilisée pour l'encoder et le décoder. Lorsqu'un type est similaire à un autre, il peut faire appel aux méthodes d'encodage et de décodage de cet autre type. Par exemple, l'encodage du type *objet* se fait par la sérialisation de l'objet à encoder, suivi d'un appel à la méthode d'encodage du type *tableau d'octets*. La procédure inverse est effectuée pour le décodage : la méthode de décodage du type *tableau d'octets* est d'abord appelée et la suite d'octets obtenue est désérialisée vers un objet.

Voici les types qui sont supportés :

- **Nombre entier (INT)** : Encodé sous forme d'un entier 32 bits avec ordre *réseau* (big-endian) pour les octets obtenus. Cette forme est donc indépendante de l'architecture matérielle.
- **Tableau d'octets (BYTE)** : Un caractère *nul* est placé à chaque 1024 octets

pour la validation.
- **Chaîne de caractères (STRING)** : Traité de la même façon que le type *tableau d'octets*.
- **Objet (OBJECT)** : Une sérialisation est effectuée sur l'objet et la suite d'octets obtenue est traitée de la même façon que le type *tableau d'octets*.

La couche de communication ne supporte que ces types, car ils répondent aux besoins des différents messages du protocole de communication du module *dstackless*. Le type *entier* est utilisé pour l'échange d'identifiants numériques ou de valeurs de comptage. Le type *chaîne de caractères* est utilisé principalement pour l'échange de noms (fonctions, classes, paramètres, etc.) et d'identifiants non numériques (*UUID*). Le type *tableau d'octets* est utilisé pour l'échange de dépendances, souvent sous la forme de fichiers binaires. Pour les paramètres dont le type est inconnu, par exemple, pour les objets envoyés sur les canaux de communication, le type *objet* est utilisé pour assurer la sérialisation nécessaire au passage sur le réseau.

Structure d'un protocole

Travaillant en étroite collaboration avec la couche de communication, on retrouve maintenant la structure elle-même du protocole. Cette structure définit un ensemble de messages supportés par celui-ci. Elle est définie à l'aide d'un dictionnaire Python, contenant tous les messages du protocole. La clé d'un élément du dictionnaire correspond au nom du message et sa valeur correspond à la configuration du message.

La configuration d'un message est un dictionnaire, qui contient deux éléments :

1. Le premier élément, identifié par la clé *ProtocolGenerator.KEY_ARGUMENT_LIST*, est la liste des paramètres du message et de leurs types. Cette liste prend la forme d'un n-uplet (*tuple* Python), qui contient l'ensemble des paramètres. Chacun des paramètre est défini à l'aide d'un n-uplet (*nom*, *type*).
2. Le deuxième élément est un pointeur vers la méthode à appeler lorsque la couche de communication a décodé le message. Cet élément est identifié par la clé *ProtocolGenerator.KEY_CALLBACK*.

Pour envoyer un message, une méthode nommée *send* est utilisée. Cette méthode reçoit l'identifiant de l'hôte distant auquel le message doit être envoyé, le nom du message à envoyer et les valeurs des paramètres du message sous forme de paramètres nommés du langage Python. Une fois appelée, la méthode *send* utilise la couche de communication pour encoder le message et l'envoyer à l'hôte distant.

Lorsqu'un message a été décodé par la couche de communication, une vérification est faite à la structure du protocole pour retrouver la méthode associée à celui-ci. Cette méthode est appelée avec les paramètres décodés et peut alors traiter le message. Ce mode de fonctionnement est événementiel. Certains événements peuvent aussi être lancés lors des connexions et des déconnexions. Lors de la définition du protocole, une surcharge des méthodes *_connect_event* et *_disconnect_event* permet de traiter ces événements.

L'exemple suivant définit les différentes parties nécessaires à la mise en place d'un protocole de communication avec le générateur de protocole :

```
# Structure du protocole
class MonProtocole(ProtocolGenerator):
    def __init__(self):
        structure = {
            # Définition d'un message.
            'MON_MESSAGE': {
                ProtocolGenerator.KEY_ARGUMENT_LIST: (
                    # Liste des paramètres du message.
                    ('param_1', ProtocolGenerator.STRING),
                    ('param_2', ProtocolGenerator.INT),
                    ('param_3', ProtocolGenerator.OBJECT)
                ),
                ProtocolGenerator.KEY_CALLBACK:
                    # Fonction à appeler lorsque ce message
                    # est décodé.
                    self.evenement_mon_message
            }
        }

        # Initialisation du générateur de protocole
        # avec le protocole défini dans la variable
        # "structure".
        ProtocolGenerator.__init__(self, structure)

    # Fonction à appeler lorsque le message "MON_MESSAGE"
    # est décodé.
    def evenement_mon_message(
                self,
                remote_host,
                param_1,
```

```
                param_2,
                param_3
            ):
        print '"MON_MESSAGE" reçu de:' \
            remote_host._remote_address, \
            remote_host._remote_port
        print param_1
        print param_2 + 100
        print param_3[2]

    def _disconnect_event(self, remote_host):
        print 'Connexion de:' remote_host._remote_address, \
            remote_host._remote_port

    def _connect_event(self, remote_host):
        print 'Déconnexion de:' remote_host._remote_address, \
            remote_host._remote_port

# Boucle de gestion des événements du protocole.
# "check_for_event" termine lorsqu'un
# message vient d'être traité.
def boucle_evenements_reseau()
    global instance_protocole

    if instance_protocole.is_listening() or \
            instance_protocole.is_connected():
        while True:
            instance_protocole.check_for_event()
            stackless.schedule()
```

Pour créer un protocole, il faut tout d'abord créer une nouvelle classe qui étend la classe *ProtocolGenerator*. Dans le constructeur de cette classe (méthode *__init__*), la structure du protocole est définie et est envoyée au constructeur de la classe parente à l'instruction **ProtocolGenerator.__init__(self, structure)**. Dans l'exemple, le protocole ne contient qu'un message, nommé *MON_MESSAGE*. Ce message a trois paramètres : *param_1* de type *STRING*, *param_2* de type *INT* et *param_3* de type *OBJECT*.

Lorsque la couche de communication a décodé un message de type *MON_MESSAGE*, la méthode *evenement_mon_mesage* est appelée. Cette méthode est définie dans la

classe du protocole et est associée au message dans la structure du protocole. C'est avec cette association que la couche de communication peut la retrouver. Le message reçu est traité à l'intérieur de la méthode *evenement_mon_message*. Dans l'exemple, il y a un affichage de la provenance du message ainsi que des paramètres sur lesquels quelques opérations sont effectuées.

Une surcharge des méthodes *_disconnect_event* et *_connect_event*, de la classe *ProtocolGenerator*, a été effectuée. Cette surcharge permet d'afficher un message lorsqu'il y a une connexion ou une déconnexion d'un hôte distant.

Finalement, la fonction *boucle_evenement_reseau* sert à demander régulièrement au protocole de lire les messages reçus. C'est à ce moment que le décodage d'un message peut survenir. Dans le cas de ce protocole, c'est à ce moment que la méthode *evenement_mon_message* sera appelée.

Pour l'utilisation de ce protocole, on doit retrouver au moins un serveur et un client. Le code suivant permet la création de la partie serveur :

```
instance_protocole = MonProtocole()
instance_protocole.listen(6473)

stackless.tasklet(boucle_evenements_reseau)()
stackless.run()
```

Une instance du protocole est tout d'abord créée. L'appel à la méthode *listen* indique ensuite à l'instance d'écouter sur un port, en attente de connexion. La fonction *boucle_evenement_reseau* est démarrée, sous forme d'un microprocessus, pour vérifier régulièrement si des messages ont été reçus.

Le code suivant correspond à la partie client, qui se connecte sur le serveur et lui envoie un message :

```
def autre_traitement(valeur):
    while True:
        print 'Valeur "autre_traitement":', valeur
        valeur = valeur + 1
        stackless.schedule()

instance_protocole = MonProtocole()

remote_host = instance_protocole.connect("127.0.0.1", 6473)
instance_protocole.send(remote_host,
```

```
                                'MON_MESSAGE',
                                param_1='Hello World!',
                                param_2=17,
                                param_3=['Hello', 'World', 17]
)

stackless.tasklet(autre_traitement)(3)
stackless.tasklet(boucle_boucle_evenements_reseau)()
stackless.run()
```

Dans cet exemple, une instance du protocole est aussi créée. Cette fois, au lieu d'écouter sur un port, le client se connecte à l'aide de la méthode *connect* sur l'adresse *IP* et sur le port sur lesquels le serveur écoute. Une fois connecté, un message de type *MON_MESSAGE* est envoyé au serveur. Du côté du serveur, le message est décodé et la méthode *evenement_mon_message* est appelé pour le traiter.

Sur une instance de l'interpréteur, la gestion des événements réseau peut se faire en concurrence avec les autres traitements en cours à l'aide des microprocessus. Ce programme client démontre la façon de faire. La fonction *boucle_evenement_reseau* est encore démarrée sous la forme d'un microprocessus, mais cette fois, son exécution est faite en alternance avec l'exécution de la fonction *autre_traitement*. Cette dernière est aussi démarrée sous la forme d'un microprocessus.

3.4.4 Résumé

En résumé, un nœud est une instance de l'interpréteur PyPy à laquelle un identifiant unique a été attribué. Une fois identifié, un nœud écoute sur un port *TCP/IP* en attente de connexions, en provenance d'autres nœuds.

La connexion entre deux nœuds survient suite à une migration de microprocessus, à la réception d'une référence vers un canal de communication partagé ou lors de la réception d'une référence vers un objet distribué. Lorsque deux nœuds sont connectés, ils ont à échanger plusieurs messages. Ces messages sont reliés à toutes les opérations possibles entre eux, par exemple, la migration de microprocessus, les échanges par les canaux de communication ou l'accès aux objets distribués. Ces messages sont définis à l'intérieur d'un protocole de communication.

Pour la création du protocole de communication, un générateur de protocole de

communication a été mis en place. La structure du protocole est définie sous forme de configuration. Cette configuration est utilisée pour l'encodage et le décodage des messages qui doivent passer par le réseau. Lorsqu'un message est décodé, une méthode associée à celui-ci est appelée, pour le traiter. La réception des messages se fait donc de façon événementielle.

3.5 Canal de communication

Avec le modèle de communication de Stackless Python, un canal de communication est un objet partagé par deux microprocessus ou plus leur permettant d'échanger des données. Un canal est représenté par un objet de type *channel*. Le microprocessus qui doit recevoir un objet appelle la méthode *receive* d'un objet canal et celui qui veut envoyer un objet appelle la méthode *send*.

La communication est faite de façon bloquante. Lorsqu'un appel est fait à une de ces méthodes, le microprocessus reste bloqué tant qu'un appel n'est pas fait à la méthode opposée. Dans le cas où plusieurs microprocessus doivent être bloqués, les requêtes sont placées dans une file d'attente appartenant au canal et sont traitées dans l'ordre où elles sont arrivées.

3.5.1 Sémantique

Pour notre implémentation des canaux de communication, la classe *dstackless.channel* n'hérite pas de celle du module *stackless* de PyPy. Le comportement de la communication locale est basé sur le même algorithme, mais une couche de communication réseau a été ajoutée. Cette nouvelle couche est utilisée lors des échanges devant se faire entre deux nœuds. Cet ajout est fortement lié au comportement de base des canaux du module *stackless*, car le canal doit maintenant gérer les requêtes locales au nœud et les requêtes en provenance de nœuds distants.

Un canal est donc à la fois local et à la fois réseau. Il s'occupe d'envoyer des notifications à tous les nœuds qui l'utilisent pour indiquer qu'il y a des requêtes en attente. Il gère aussi la synchronisation entre les nœuds pour l'envoi des objets devant être transférés par le réseau. Une liste contenant les identifiants uniques des nœuds utilisant un canal permet de limiter l'envoi des notifications à ceux-ci. Ce modèle hybride simplifie grandement la migration de microprocessus, car il n'est pas nécessaire de convertir un

canal local en canal réseau et vice-versa. De plus, ce modèle hybride est inévitable, car plus de deux microprocessus peuvent partager un même canal, et seul l'un d'entre eux pourrait se trouver sur un nœud différent.

Une différence existe entre une communication locale à un nœud et distante entre deux nœuds. Dans le premier cas, le transfert d'un objet a lieu par référence, de la même façon qu'avec les canaux standards du module *stackless*. Dans le second cas, ce passage a lieu par copie étant donné que des étapes de sérialisation et de désérialisation sont nécessaires pour transférer un objet via le réseau. La sérialisation est gérée de façon transparente, à l'aide du type *objet* du protocole de communication entre les nœuds. Lorsqu'un comportement spécifique est nécessaire pour la sérialisation de certains objets, il peut être mis en place à l'aide des surcharges offertes par le module *pickle*.

Pour la communication à distance, le mode de fonctionnement des canaux est de type *pair-à-pair décentralisé* [34]. De cette façon, l'utilisation d'un serveur de noms n'est pas nécessaire. Dans un objet canal, une liste des nœuds utilisant le canal est maintenue. Chacun des nœuds présents dans cette liste doit avoir une connexion *TCP/IP* directe vers les autres nœuds, aussi présents dans celle-ci. Une copie de l'objet canal existe par contre sur chacun des nœuds utilisant un même canal. Ils ont tous la responsabilité de mettre à jour la liste des utilisateurs du canal, présente dans leur propre copie de l'objet canal, lorsqu'il sont informés de l'ajout ou de la disparition d'un nœud utilisant le canal. Les étapes suivantes surviennent lorsqu'un nœud reçoit un objet canal :

1. L'objet canal est d'abord désérialisé. Le processus de désérialisation s'occupe des autres étapes.
2. Un parcours de la liste des identifiants uniques des nœuds utilisateurs, présente dans la copie de l'objet canal, est effectué.
3. Le nœud établit alors une connexion vers tous les nœuds avec lesquels il n'est pas déjà connecté.
4. Le nœud envoie ensuite une notification à tous les nœuds, présents dans la liste, pour leur indiquer qu'il utilise maintenant le canal.
5. Les autres nœuds l'ajoutent alors dans leur propre liste des utilisateurs du canal.

Une forme simple de ménage est mise en place pour nettoyer les listes d'utilisateurs des canaux. Deux cas peuvent survenir :

1. Le premier cas survient lorsqu'un nœud A détecte la déconnexion d'un nœud B. Il fait alors le tour de tous ses canaux pour retirer B de leurs listes de nœuds utilisateurs.

2. Le cas où un nœud A n'a plus aucun microprocessus qui utilise un canal est géré à l'aide d'un dictionnaire à référence faible. Une fois toutes les références fortes vers le canal supprimées, celui-ci est automatiquement supprimé de ce dictionnaire par le ramasse-miettes de l'interpréteur. Si A reçoit une notification pour ce canal, il est alors en mesure de savoir qu'il ne l'utilise plus et il peut en informer les autres nœuds. Ceux-ci doivent alors retirer A de leur copie de la liste des utilisateurs du canal. Suite à ce retrait, A ne recevra plus aucune notification pour le canal qu'il n'utilise plus.

3.5.2 Protocole d'échange entre les nœuds

L'interaction nécessaire entre les nœuds utilisateurs d'un canal de communication distribué implique l'échange de plusieurs messages. Ces échanges servent à la synchronisation, lorsque des requêtes sont en attente, ainsi qu'au transfert des objets entre les nœuds. Les différents messages nécessaires au fonctionnement des canaux de communication font partie du protocole de communication mis en place au niveau du nœud.

Le premier message, *CHANNEL*, est envoyé par un nœud, aux autres nœuds présents dans la liste des utilisateurs d'un canal, pour les informer qu'il est lui aussi un utilisateur du canal. Cette notification est envoyée lors de la désérialisation d'un objet canal que le nœud vient de recevoir. Ce message sert donc à établir le réseau d'utilisateurs pour un canal.

Une fois ce réseau établi, les notifications liées aux requêtes d'envoi et de réception sur le canal peuvent être envoyées. Le modèle de communication choisi autorise l'écriture, ainsi que la lecture, par plusieurs microprocessus. Si une requête ne peut immédiatement être complétée, elle est placée dans une file d'attente. À ce moment, les autres nœuds utilisateurs en sont informés. Une notification leur est aussi envoyée lorsqu'une requête, qui se trouvait dans la file d'attente, a été complétée.

Le message *CHANNEL_REQUEST* est utilisé pour informer les autres nœuds du nombre de requêtes en attente sur le canal. Dans la file d'attente d'un canal, il ne peut y avoir à la fois des requêtes d'envoi et des requêtes de réception, car lorsque ce cas survient, la communication peut être complétée immédiatement. Les requêtes doivent donc être du même type pour être placées dans la file d'attente. Le signe de la valeur envoyée avec le message *CHANNEL_REQUEST* indique le type des requêtes en attente. Lorsque la valeur est positive, elle indique le nombre de requêtes d'envoi en attente. Lorsqu'elle est négative, elle indique le nombre de requêtes de réception en attente.

À la réception d'un message de type *CHANNEL_REQUEST*, le nœud qui reçoit cette notification peut alors répondre immédiatement à une requête en attente ou l'insérer en tant que requête distante dans la file d'attente de son objet canal. Une telle requête peut être retirée de la file d'attente lors de la réception d'un message de type *CHANNEL_REQUEST*, qui indique qu'elle a été répondue par un autre nœud. Ce cas survient lorsque la file d'attente de l'objet canal contient plus de requêtes, reliées au nœud qui a envoyé le message, que ce que le message indique.

Une requête distante placée dans la file d'attente d'un canal est gérée à l'aide d'un microprocessus. L'échange entre ce microprocessus temporaire et celui du programme, qui répond à la requête de communication, se fait de la même façon qu'une communication qui aurait lieu entre deux microprocessus exécutés sur un même nœud. Par contre, le microprocessus temporaire s'occupe d'envoyer la donnée au nœud ayant amorcé la requête, dans le cas d'une requête de lecture, ou de récupérer cette donnée, dans le cas d'une requête d'écriture. Une fois l'échange terminé et validé, le rôle du microprocessus temporaire se termine. Si une requête est annulée et qu'elle est retirée de la file d'attente, l'exécution de ce microprocessus termine aussi.

L'échange d'un objet entre deux nœuds se fait à l'aide de cinq messages. Ces messages servent à synchroniser le transfert de l'objet, vérifier qu'il a bien été transféré et s'assurer que deux nœuds ne répondent pas en même temps à une même requête. La séquence des messages échangés est différente selon le type de la requête (envoi ou réception) :

1. Le diagramme de flux présenté à la figure 3.2 décrit le cas où une requête de réception locale doit être répondue par une requête d'envoi distante. La requête de réception locale génère l'envoi du message *GET_DATA*, pour demander l'objet. Le nœud distant peut alors répondre par le message *DATA*, pour retourner la l'objet demandé, ou *NO_DATA*, pour indiquer que la requête d'envoi n'existe plus.
2. La figure 3.3 présente le deuxième cas, où une requête d'envoi locale doit répondre à une requête de réception à distance. La requête d'envoi locale génère l'envoi du message *DATA*, pour envoyer l'objet directement. Le nœud distant peut alors répondre par le message *RECV*, pour indiquer que l'objet a bien été reçu, ou *NO_RECV*, pour indiquer que la requête de réception n'existe plus et que l'objet n'a pas été reçu.

Certaines améliorations peuvent être apportées à ce protocole. Par exemple, dans le cas de la réponse à une requête de réception distante, l'envoi direct de l'objet cause un transfert inutile, lorsque la requête de réception n'est plus disponible. Cela peut causer des problèmes de performance lors de l'envoi d'objets de taille importante. Par

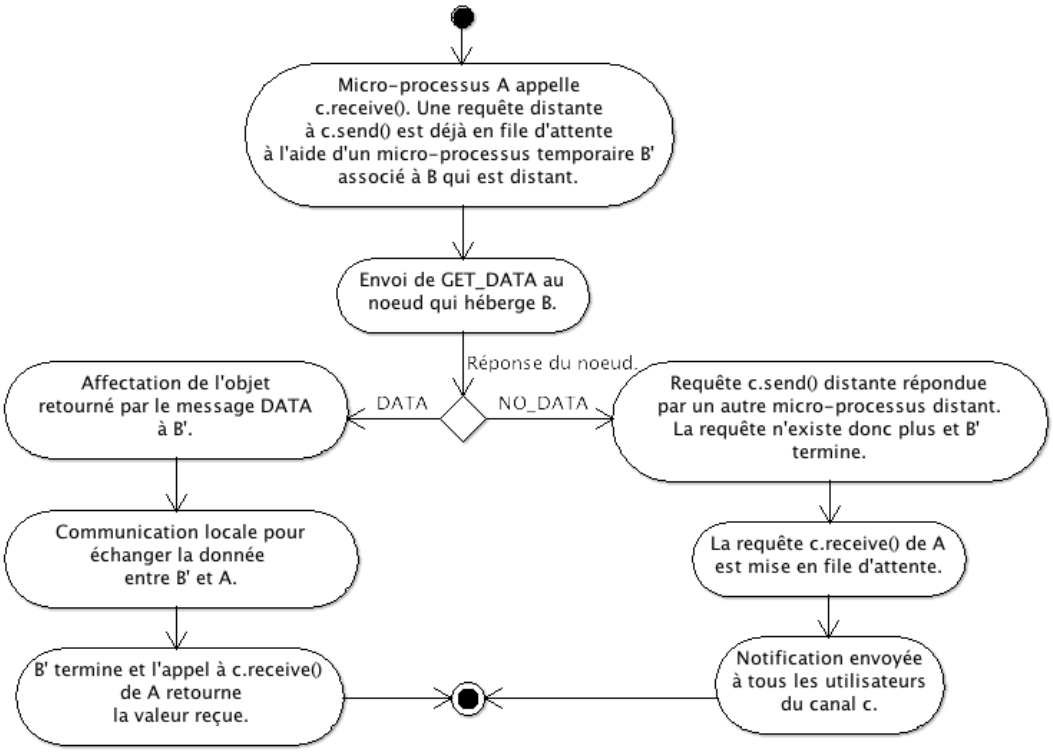

FIGURE 3.2 – Requête de réception avec requête d'envoi distante.

contre, le fonctionnement actuel répond aux besoins de la première version du module *dstackless*.

3.6 Dépendances

La méthode choisie pour la gestion des dépendances est une récupération tardive, basée sur le système de sérialisation de Python. Lors de la sérialisation, celui-ci note automatiquement le module de chacun des objets sauvegardés. Pendant l'étape de désérialisation, un appel à la fonction *__import__* est alors effectué pour recharger les modules associés aux objets à récupérer et qui ne sont pas présentement en mémoire.

Lorsque le programmeur veut recharger un objet dans une instance fraîche de l'interpréteur, il n'a donc pas à se soucier du rechargement du module associé à l'objet. Lors de la migration d'un microprocessus, les identifiants des modules associés aux objets, sauvegardés avec celui-ci, sont donc sauvegardés. Au besoin, ces modules sont rechar-

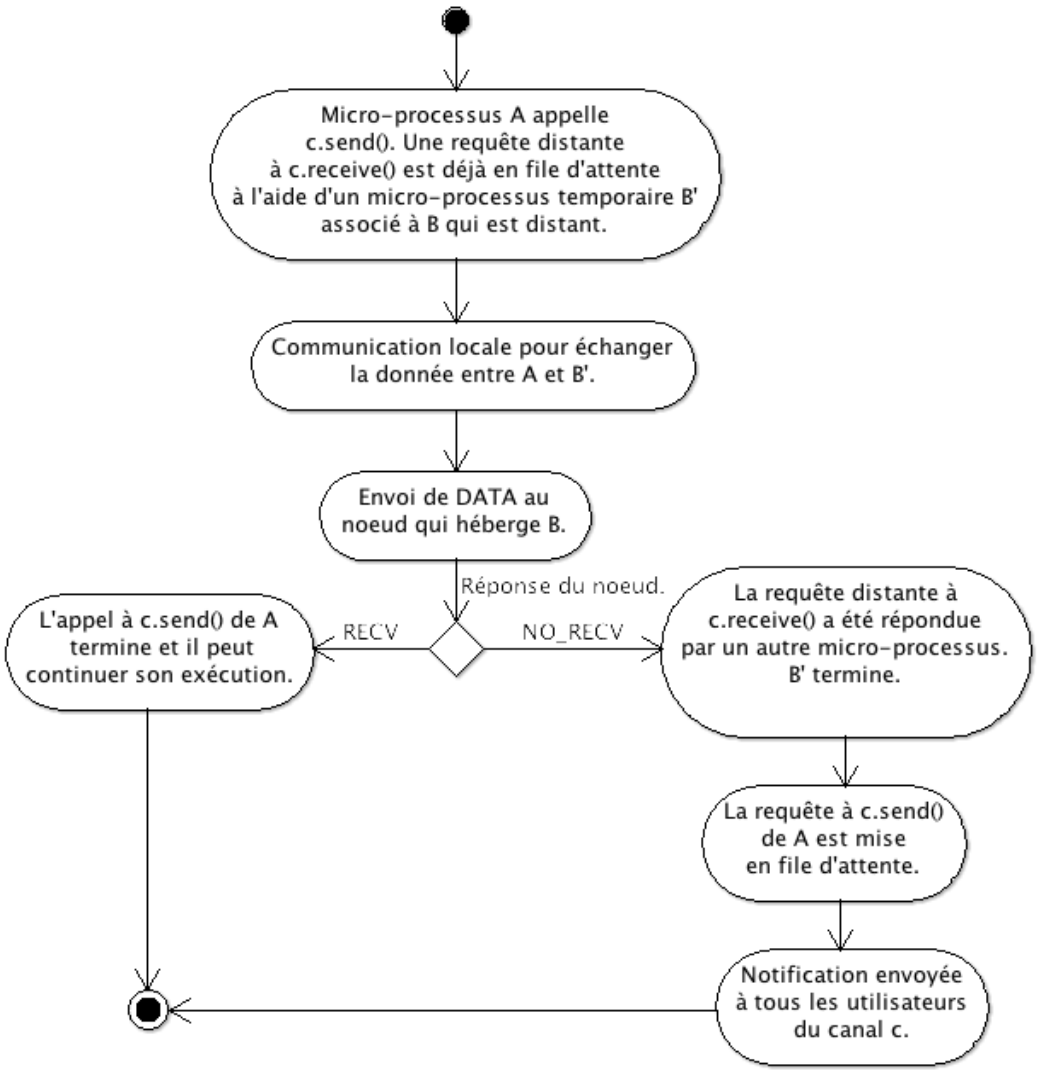

FIGURE 3.3 – Requête d'envoi avec requête de réception distante.

gés automatiquement sur le nœud récepteur. Tout ce processus ne nécessite aucune intervention du programmeur.

Système d'importation de Python

L'importation d'un module Python se fait en deux étapes pouvant être répétées plusieurs fois dans le cas du chargement d'un sous-module :

1. La première étape est la recherche du module demandé dans les répertoires connus

par l'interpréteur. Ces répertoires sont ceux contenus dans la liste associée à la variable *sys.path*. Au démarrage de l'interpréteur, cette variable contient les répertoires de toutes les bibliothèques de base de l'interpréteur Python, ainsi que ceux présents dans la variable d'environnement *PYTHON_PATHS*.

2. La deuxième étape est le chargement du module en mémoire, s'il a été trouvé lors de la première étape. Lorsque l'importation d'un sous-module est demandée, chacun des modules parents est retrouvé et chargé par ce mécanisme, avant que le sous-module soit chargé.

L'utilisation du mécanisme de base se complique lorsqu'un nœud désérialise un objet pour lequel le module n'est pas disponible. Cet objet peut avoir été reçu à même l'état d'exécution d'un microprocessus ou via un canal de communication. Dans ce cas, l'interpréteur lève une exception de type *ImportError*, car il ne trouve pas le module. La désérialisation termine alors, empêchant le rechargement du microprocessus ou de l'objet.

Il serait possible d'intercepter cette exception et de recommencer l'opération ayant échoué ; cependant, l'interpréteur du langage Python offre un meilleur outil : un crochet (*hook* en anglais), à même le système d'importation [47], permet d'en modifier le comportement. Tous les appels aux mots-clés *import* et *from [paquet] import [nom]*, et à la fonction *__import__*, sont alors interceptés.

Lors de l'utilisation du crochet du système d'importation, il est tout de même possible de faire directement appel au système de base, à l'aide module *imp*. L'exemple qui suit démontre l'utilisation de ce crochet, pour la mise en place d'un système d'importation personnalisé :

```
class MetaImporter(object):
    _module_ref = {}

    def find_module(self, fullname, path=None):
        # Retrouver le module et noter ses informations
        # (fichier, chemin, description).
        print "Recherche du module: ", fullname

        self._module_ref[fullname] = imp.find_module(
            fullname,
            path
        )

        # Retourner un chargeur qui
```

```
            # implémente load_module().
            return self

        def load_module(self, fullname):
            # Charger le module.
            print "Chargement du module: ", fullname

            module_ref = self._module_ref[fullname]
            module = imp.load_module(fullname, *module_ref)

            # Fermer le fichier Python associé au module.
            module_ref[0].close()
            self._module_ref.pop(fullname)

            return module

    sys.meta_path = [MetaImporter()]
```

Dans cet exemple, la classe *MetaImporter* correspond à un système d'importation personnalisé. Un tel système doit remplir deux fonctions, correspondant aux deux étapes du système d'importation de base :

1. La première étape est effectuée à l'aide de la méthode *find_module*, qui sert à retrouver le module à charger. Dans l'exemple, on affiche que l'on est à la recherche d'un module et on fait appel au système d'importation de base pour retrouver le module. Cette méthode doit retourner un objet permettant de charger le module retrouvé, ou *None*, lorsque le module n'a pas été trouvé.
2. La deuxième étape est effectuée à l'aide de l'objet retourné par l'appel à *find_module*. Un appel à la méthode *load_module* de cet objet est effectué. Dans cette méthode, le module doit être chargé et retourné. Dans l'exemple, on affiche que l'on est en train de charger le module et on fait ensuite appel au système d'importation de base pour charger le module.

Pour être utilisé, un système d'importation personnalisé doit être ajouté à la variable *sys.meta_path*. Cette variable correspond au crochet. Elle doit pointer vers une liste qui contient chacune des instances des systèmes d'importation personnalisés. Si un appel à la méthode *find_module* d'un système d'importation personnalisé retourne *None*, un appel sera effectué au prochain système présent dans la liste, jusqu'à ce qu'un des appels retourne l'objet permettant de charger le module. Si tous les appels retournent *None*, un appel au système de base est effectué. Dans l'exemple précédent, le crochet

ne contient qu'un seul système personnalisé. Ce système correspond à une instance de la classe *MetaImporter*.

Récupération des dépendances

Une récupération de dépendance peut se produire lors de la désérialisation d'un objet reçu d'un autre nœud. Cette récupération est faite à même la méthode *find_module* d'un système d'importation personnalisé. Un appel est tout d'abord effectué au système d'importation de base, à l'aide de *imp.find_module*. Si cet appel lance une exception de type *ImportError*, le module n'est pas disponible. C'est à ce moment qu'une requête de récupération est envoyée nœud ayant envoyé l'objet à désérialiser.

La désérialisation d'un objet est effectuée à l'intérieur d'un microprocessus temporaire. Une fois la requête de récupération envoyée, le microprocessus temporaire est bloqué. Suite à la réception de la réponse, le module reçu est sauvegardé dans un répertoire spécifique au module *dstackless*, mais connu du système d'importation de base. Ce répertoire, en plus d'être utilisé pour la récupération des dépendances, sert aussi de mémoire cache pour les chargements futurs. Le microprocessus temporaire est finalement débloqué et le système d'importation de base est appelé à nouveau pour charger le module qui est maintenant disponible.

Il est important d'effectuer toutes les opérations de désérialisation dans des microprocessus temporaires, afin de ne pas bloquer un microprocessus important au bon fonctionnement du module *dstackless*. Le meilleur exemple d'un microprocessus jugé important est celui qui s'occupe de la boucle d'événements réseau. Si ce dernier était bloqué, le protocole d'échange avec les autres nœuds cesserait de fonctionner et la réponse à la requête de récupération d'un module ne serait jamais reçue.

Le même système est utilisé lors des appels subséquents au mot-clé *import*, effectués par un microprocessus qui a été migré au moins une fois. Lorsque le nœud sur lequel le microprocessus a été migré ne connaît pas le module demandé, il peut demander le module au nœud sur lequel le microprocessus était initialement exécuté. L'avantage de cette récupération tardive est que seul le code réellement nécessaire est copié pour continuer l'exécution d'un microprocessus. Si, dans le futur, il a besoin d'un nouveau module, il ne sera récupéré qu'à ce moment. Par contre, si le nœud d'origine est arrêté, cette récupération tardive ne pourra avoir lieu et une exception de type *ImportError* devra être levée.

Le choix du nœud auquel on doit demander un module dépend de l'événement qui

initie une demande d'importation de modules, et du type d'objet qui est lié à cette demande :

1. Le premier cas est une demande d'importation liée à un microprocessus. Il survient suite à une opération de migration, lors de son rechargement, ou lorsque le microprocessus effectue une demande d'importation d'un module par une utilisation du mot-clé *import*. La requête de récupération est effectuée auprès du nœud d'origine du microprocessus. L'identifiant de ce nœud a été associé au microprocessus lors de sa création.
2. Le dernier cas survient lors de la désérialisation d'un objet, reçu par un canal de communication. La requête pour récupérer le module est alors effectuée auprès du nœud ayant envoyé l'objet à désérialiser.

3.7 Variables globales et objets partagés

Comme la plupart des langages objets, le langage Python n'offre qu'un accès par référence à un objet associé à une variable. Lorsque cette variable est envoyée en tant que paramètre pour l'appel d'une fonction ou d'une méthode, seule la référence est transmise. À l'exception des objets immuables, tels les entiers, les chaînes de caractères ou les tuples, toute mise à jour d'un objet reçu en paramètre d'une fonction sera visible pour l'appelant de cette fonction.

Le même comportement est mis en place pour les canaux de communication lors des échanges locaux. Seule la référence vers l'objet est échangée ce qui permet à plusieurs microprocessus de partager un même objet. Ce partage d'objets mutables risque de poser problème lorsque ces objets sont échangés entre plusieurs nœuds, car cet échange se fera alors par copie compte tenu de l'étape de sérialisation et de désérialisation nécessaire pour le passage par le réseau. Voici les détails de l'implémentation des méthodes, énoncées dans le chapitre précédent, pour aider le programmeur à éviter les problèmes liés au partage d'objets.

3.7.1 Environnement global

Lors du chargement d'un module Python, un dictionnaire vide est associé à ce dernier. Il sert d'environnement global pour le module auquel il est associé. L'analyse et l'exécution du fichier source, faisant partie du chargement, remplissent alors ce dictionnaire avec chacune des variables pour lesquelles il y a eu une assignation, ainsi que

chacune des fonctions et classes déclarées.

Basé sur le principe de module, le module *dstackless* pousse un peu plus leur utilisation. Pour éviter le partage d'un environnement global entre plusieurs microprocessus, un nouveau module est créé pour chacun d'eux. Ce module est créé à partir du fichier source dans lequel se trouve la fonction utilisée comme point d'entrée du microprocessus, mais porte comme nom le *UUID* du microprocessus auquel il est associé. Plusieurs microprocessus créés à partir d'une même fonction créent donc autant de modules distincts. Chacun de ces modules a son propre dictionnaire global. Ce dictionnaire n'est donc pas partagé et peut facilement suivre le microprocessus auquel il est associé lors d'une migration.

Contrairement à la duplication de processus des systèmes *POSIX*, aucune copie d'environnement global n'est effectuée. Chacun des microprocessus est créé comme s'il était une nouvelle instance d'un programme. Avec ce comportement, il n'y a alors que deux moyens pour partager un canal de communications entre plusieurs microprocessus :

- Le passer en paramètres à la fonction utilisée comme point d'entrée de chacun des microprocessus.
- L'échanger par une communication sur un autre canal de communication partagé.

Les variables globales peuvent tout de même être utilisées, mais le partage est limité aux fonctions et méthodes appelées par un même microprocessus et faisant partie du même module que celui-ci. De toute façon, ces fonctions et méthodes font partie intégrante de l'environnement du microprocessus et seront migrées avec celui-ci.

3.7.2 Objets distribués

Pour les canaux de communication, nous avons choisi d'effectuer la communication par référence lors des échanges locaux à un nœud et par copie lors des échanges entre deux nœuds. Ces choix sont motivés par des raisons de performance, pour éviter d'avoir plusieurs copies locales de gros objets, et techniques, car deux nœuds ne partagent pas la même mémoire et l'encodage nécessaire pour que les objets soient transférés par le réseau implique une copie de ceux-ci. Ce comportement par défaut n'est pas toujours approprié, car on peut avoir besoin de transférer des objets par référence, entre deux nœuds, lorsqu'ils sont trop volumineux pour que la copie soit complétée dans un délai raisonnable.

Pour la communication, compte tenu des détails d'implémentation de l'interpréteur Python, il n'est pas trivial de donner le choix du comportement par défaut au program-

meur. Il est impossible d'associer de nouvelles méthodes aux types de base fournis par le langage sans modifier le code de l'interpréteur. Cette limitation empêche la création de comportements personnalisés pour l'accès aux objets. Il est alors difficile de mettre en place un système d'objets distribués, où seule une référence est échangée entre les nœuds. Le sous-classement des types peut sembler une solution, mais il implique que le programmeur ne peut plus utiliser les types de base du langage lorsqu'il veut que des objets soient envoyés à distance par référence.

Objets mandataires de PyPy

La solution qui semble la plus appropriée et qui a été mise en place est basée sur le modèle de gestion des objets par objet mandataire [24], spécifique à PyPy. Ce modèle de gestion permet de prendre un objet de base et de lui associer une fonction, qui est appelée lors de tout accès à l'objet. Cette association se fait par la création d'un objet mandataire se présentant au programmeur comme s'il était l'objet original ; son type est aussi le même. Voici un exemple, où toutes les opérations passent par la fonction de l'objet mandataire pour être affichées et sont par la suite déléguées à l'objet original :

```
from tputil import make_proxy

def afficheur(operation):
    print "Appel de:", operation.opname
    print "Arguments de l'appel:", operation.args
    print "ID de l'objet original:", id(operation.obj)
    print "ID de l'objet mandataire:", id(operation.proxyobj)
    operation.delegate()

o = []
# Affichage de certaines informations
# sur l'objet original.
print "ID de l'objet:", id(o)
print "Type de l'objet:", type(o)

# Création de l'objet mandataire.
l = make_proxy(afficheur, obj=o)

# Affichage de certaines informations
# sur l'objet mandataire.
print "ID de l'objet mandataire:", id(l)
print "Type de l'objet mandataire:", type(l)
```

```
# Appel d'une méthode sur l'objet
# mandataire. Cette méthode est
# une méthode de l'objet original.
l.append(3)
```

Une trace d'exécution démontre ce qui est affiché :

```
ID de l'objet: 2
Type de l'objet: <type 'list'>
ID de l'objet mandataire: 6
Type de l'objet mandataire: <type 'list'>
Appel de: __getattribute__
Arguments de l'appel: ('append',)
ID de l'objet original: 2
ID de l'objet mandataire: 6
Appel de: append
Arguments de l'appel: (3,)
ID de l'objet original: 2
ID de l'objet mandataire: 6
```

Création d'objets distribués

Pour répondre au besoin d'avoir des objets distribués et transmis par référence lors des communications, une méthode simple est proposée. Une fonction permet de marquer des objets comme étant distribués. Elle reçoit en paramètre l'objet que l'on veut rendre distribué et en retour elle renvoie un nouvel objet référence (objet mandataire de PyPy). Ce dernier s'utilise de la même façon que l'objet original, ayant le même type.

Toutes les opérations réalisées sur l'objet référence sont donc interceptées par une fonction intermédiaire, qui s'occupe de les traiter. Pour associer un objet mandataire à son objet original, un *UUID* est utilisé. Si l'objet original se trouve sur le même nœud que l'objet référence, l'opération lui est directement déléguée. S'il réside sur un nœud distant, l'opération et le *UUID* de l'objet sont envoyés au nœud qui l'héberge. Ce nœud utilise alors le *UUID* pour retrouver l'objet dans un dictionnaire et il s'occupe ensuite d'effectuer l'opération sur celui-ci. La réponse (valeur de retour, exception, etc.) est par la suite renvoyée au nœud qui a demandé l'opération.

Tous les détails de la fonction *make_proxy* de PyPy sont alors masqués au pro-

grammeur à l'aide de la nouvelle fonction *dstackless.ref_object*. Voici un exemple de conversion d'un objet en objet distribué et de l'envoi de sa référence à un autre microprocessus par un canal de communication :

```
def a(canal):
    # Création d'une liste vide.
    liste = []

    # Transformation en liste distribuée.
    liste_distribuee = dstackless.ref_object(liste)
    liste_distribuee.append(3)

    # Envoi de la référence à un autre microprocessus.
    canal.send(liste_distribuee)
```

Composition des objets distribués

On le sait déjà, dans le langage Python, toute variable est passée par référence, que ce soit lorsqu'utilisée comme paramètre d'une fonction ou lorsqu'utilisée comme valeur de retour. Ce comportement est tout aussi vrai pour l'accès aux méthodes et attributs d'un objet. Même si l'accès à un objet distribué A passe par son objet référence, lorsqu'une opération sur celui-ci retourne un objet B, le composant, l'accès à cet objet retourné ne sera pas intercepté par l'objet référence de A. Toute opération sur l'objet B est alors effectuée directement sur celui-ci.

Lorsqu'un objet référence est utilisé sur un nœud différent de celui sur lequel l'objet distribué associé se trouve, un objet enfant retourné suite à une opération sur l'objet référence doit être copié vers le nœud ayant effectué l'opération. Les opérations sur la copie de cet objet enfant ne se répercutent alors pas sur la copie d'origine, comme le programmeur pourrait s'y attendre. Le système d'objets distribués doit donc tenir compte de ce cas. La solution est très simple. Lorsqu'une opération sur un objet distribué retourne une valeur, un objet référence pointant vers cette valeur doit être retourné à la place de celle-ci. De cette façon, on s'assure que toutes les opérations effectuées sur un objet distribué sont faites à l'aide d'objets références et que ces opérations ne génèrent pas de copies des objets le composant.

Accès à distance à un objet distribué

Avec Python, toute requête à un objet, que ce soit l'accès à un attribut ou l'appel à une méthode, demande tout d'abord un appel à la méthode *__getattribute__* pour retrouver l'attribut ou la méthode en question. Dans le cas de l'accès à un attribut, *__getattribute__* retourne directement l'attribut. Dans le cas de l'appel d'une méthode, *__getattribute__* retourne une référence vers la méthode, qui est par la suite appelée. Lorsque *__getattribute__* ne retrouve pas l'objet ou la méthode, une exception de type *AttributeError* est levée.

Pour les requêtes effectuées à distance sur un objet distribué, l'appel à *__getattribute__* ainsi que l'appel à une méthode doivent être redirigés vers le nœud qui héberge l'objet. Lors d'une telle requête, le microprocessus qui effectue la requête doit être bloqué, en attente de la réponse du nœud distant. Ce blocage se fait de la même manière qu'avec les canaux de communication, en retirant le microprocessus de la file d'attente de l'ordonnanceur. Il existe trois types de réponses à une requête effectuée à distance :

1. Le premier est le retour d'un objet. Cet objet est soit un attribut de l'objet distribué, retrouvé par l'appel à *__getattribute__*, soit la valeur de retour de l'appel d'une méthode. L'objet retourné est transformé en objet distribué et seule une référence est retournée. Les opérations sur cet objet sont aussi effectuées à distance.
2. Le deuxième est un retour d'une méthode. Lorsqu'on veut appeler une méthode sur un objet distribué, une requête est tout d'abord effectuée à la méthode *__getattribute__* de l'objet référence, pour demander si la méthode existe, à l'aide de son nom. Cette requête est envoyée au nœud hébergeur. Si la méthode existe, une référence vers celle-ci est retournée, sinon une exception *AttributeError* est levée. Lorsqu'une référence est retournée, seul le nom de la méthode est envoyé au nœud demandeur, sous forme de *réponse méthode*. Du côté du nœud ayant effectué la requête à distance, ce nom est alors utilisé pour créer une fermeture Python, qui est appelée. Cet appel génère une nouvelle requête à distance pour maintenant appeler la méthode sur l'objet distribué.
3. Le dernier type de retour est lorsqu'une exception est levée. Cette exception est renvoyée au nœud ayant effectué la requête à distance et être levée à cet endroit.

3.7.3 Migration d'objets distribués

Au même titre que les microprocessus, les objets distribués peuvent être migrés d'un nœud à un autre. Encore une fois, lorsqu'un objet est transformé en objet distribué,

une méthode *move* lui est ajoutée et c'est cette dernière qui doit être appelée pour migrer l'objet. Lors de la migration d'un objet distribué, tous les nœuds ayant un objet référence pointant vers celui-ci sont alors informés de l'identifiant unique du nouveau nœud l'hébergeant. Par contre, dans le cas des objets distribués créés à partir d'objets d'entrées/sorties, tels les fichiers où les connexions réseau, une migration ne peut avoir lieu, car ils sont fortement liés au systèmes sur lequel ils se trouvent. Voici un exemple de migration d'un objet distribué :

```
def a(canal):
    # Création d'une liste distribuée.
    liste_distribuee = dstackless.ref_object([])
    lliste_distribuee.append(3)

    # Envoi de la référence à un autre microprocessus.
    canal.send(liste_distribuee)

    # Migration de la liste distribuée vers un autre nœud.
    liste_distribuee.move(("127.0.0.1", 6473))
```

Dans cet exemple, une liste distribuée est créée. La référence est tout d'abord envoyée à un autre microprocessus, via le canal de communication *canal*. Suite à cet envoi, la liste est migrée vers un autre nœud.

L'objet référence retourné à la place d'un objet enfant, suite à une opération sur un objet distribué, n'est pas créé tout à fait de la même façon que lorsque le programmeur crée lui même un objet distribué. Cet objet référence contient le *UUID* de l'objet distribué parent auquel l'objet enfant est associé. Le même processus est appliqué lorsqu'une opération effectuée sur l'objet référence de l'objet enfant retourne aussi un de ses enfants. Un objet référence est aussi retourné, mais celui-ci contient le *UUID* de l'objet parent de plus haut niveau (objet distribué créé par le programmeur).

Lors de la migration d'un objet distribué vers un autre nœud, le *UUID* que les objets références vers ses objets enfant contient devient utile. Lorsque qu'un objet distribué est migré, tous les objets le composant sont aussi migrés avec celui-ci. Les références vers cet objet distribué, ainsi que toutes celles vers ses objets enfants, doivent être mises à jour pour indiquer où ils ont été déplacés. Lorsque les autres nœuds sont informés de la migration d'un objet, le *UUID* est utilisé pour mettre à jour la référence vers cet objet, ainsi que les références vers ses objets enfants. Les requêtes sur ces objets références seront alors envoyées vers le nouveau nœud qui les héberge.

3.7.4 Ramasse-miettes distribué

La mise en place d'un ramasse-miettes distribué se base sur une certaine coopération entre les différents nœuds utilisateurs d'un objet distribué et le nœud qui l'héberge. Ce dernier s'occupe de son côté de la coopération nécessaire avec le ramasse-miette de l'interpréteur, qui a le rôle final dans la destruction de l'objet distribué. Il faut donc un mécanisme pour s'assurer que l'objet ne soit pas détruit, tant et aussi longtemps qu'il y a encore des références à distance vers celui-ci, et ce, même s'il n'existe plus aucune référence locale.

Fonctionnement au niveau du nœud hébergeur

Au niveau du nœud qui héberge un objet distribué, le ramasse-miettes de l'interpréteur s'occupe de le supprimer dès qu'il n'y a plus de référence forte vers celui-ci. Si un nœud distant utilise l'objet, au moins une référence forte doit exister vers celui-ci, tant et aussi longtemps que le nœud distant en a besoin.

Pour répondre à ce problème, un objet intermédiaire est mis en place entre l'objet distribué et le nœud distant qui l'utilise. Cet objet intermédiaire est créé automatiquement lorsqu'une demande de sérialisation est faite, par exemple, lors de l'envoi de l'objet distribué par un canal de communication. Lorsque l'objet intermédiaire est créé, un *UUID* lui est associé et est utilisé comme clé pour le stocker dans un dictionnaire. Ce *UUID* est aussi utilisé par les nœuds qui utilisent l'objet distribué pour effectuer des requêtes à distance sur celui-ci.

Deux rôles sont remplis par l'objet intermédiaire :

1. Le premier est de recevoir les requêtes des nœuds distants et les rediriger vers l'objet qui doit les traiter.
2. Le second est de tenir une liste de tous les nœuds distants utilisant l'objet distribué, ainsi que la date et l'heure de la dernière requête sur celui-ci.

Un microprocessus s'occupe de faire le ménage du dictionnaire contenant les objets intermédiaires toutes les 120 secondes. Si un objet intermédiaire n'a reçu aucune requête d'un nœud présent dans sa liste des utilisateurs de l'objet distribué depuis plus de 120 secondes, l'identifiant du nœud est retiré de cette liste. S'il n'y a plus aucun nœud dans la liste d'un objet intermédiaire, ce dernier est retiré du dictionnaire, car il n'y a plus aucune référence à distance. Le ramasse-miettes de l'interpréteur peut alors faire son

travail et supprimer l'objet distribué, s'il n'y a plus aucun microprocessus du même nœud qui y réfère.

Fonctionnement au niveau d'un nœud utilisateur

La structure est semblable pour les nœuds qui utilisent à distance des objets distribués. Pour ces nœuds, un objet référence est créé. Cet objet a le même type que l'objet distribué, mais ne contient aucune donnée. Seuls l'identifiant unique, généré par l'objet intermédiaire associé à l'objet distribué, ainsi que l'identifiant unique du nœud hébergeur sont stockés dans l'objet référence. L'objet référence créé sur le nœud distant est aussi un objet mandataire de PyPy. Cet objet s'occupe de traiter, à l'aide d'une fonction, toutes les requêtes qui sont effectuées sur celui-ci ; ces requêtes sont alors renvoyées à l'objet intermédiaire du nœud hébergeur, qui s'occupe de les envoyer à l'objet distribué.

Tout comme pour le nœud hébergeur, un dictionnaire contient la liste de tous les objets référence. Le *UUID* de l'objet distribué est encore une fois utilisé comme clé pour référencer les objets référence. Un microprocessus parcourt la liste des références à chaque 60 secondes et informe les nœuds hébergeurs que les objets distribués sont encore utilisés. Contrairement au nœud hébergeur qui utilise un dictionnaire Python standard, un nœud distant utilise un dictionnaire à référence faible.

Lorsqu'un objet distribué n'est plus utilisé par aucun des microprocessus d'un nœud, il ne reste alors que la référence présente dans le dictionnaire à référence faible. L'interpréteur considère un objet inutilisé lorsque la seule référence restante vers celui-ci est celle d'un tel dictionnaire. Dans ce cas, le ramasse-miette de l'interpréteur supprime immédiatement l'objet. Le microprocessus exécuté toutes les 60 secondes ne voit alors plus l'objet référence et cesse d'informer le nœud hébergeur de son utilisation de l'objet distribué. Le processus de nettoyage du nœud hébergeur peut alors supprimer le nœud distant de la liste des utilisateurs de l'objet distribué.

3.8 Entrées/Sorties

Étant donné que plusieurs microprocessus peuvent être exécutés sur plusieurs noeuds, répartis sur plusieurs ordinateurs, il est important d'avoir en place un système permettant au programmeur de décider comment les entrées (STDIN) et les sorties (STDOUT)

devront être gérées. Il pourrait alors décider, par exemple, de diriger toutes les sorties vers un seul noeud qui s'occupera de les afficher, ou même de les sauvegarder dans un fichier journal sur chacun des noeuds.

3.8.1 Interception des entrées/sorties avec Python

À l'instar de plusieurs langages, Python offre des mécanismes pour intercepter ces entrées et sorties. La procédure est simple. Il suffit de remplacer les variables *sys.stdin*, *sys.stdout* et *sys.stderr* [27] par des objets qui définissent le comportement voulu. Tous les appels au mot-clé *print* et aux fonctions *input* et *raw_input* utilisent ces variables. Lorsqu'elles sont remplacées par des objets personnalisés, ce sont ces objets qui sont utilisés à la place des objets de base. L'exemple suivant montre comment rediriger la sortie standard vers un fichier.

```
import sys

f = open('journal.txt', 'w')
sys.stdout = f

print "Ce message est sauvegardé dans un fichier"
```

Cet exemple se base sur le fait que la sortie standard est utilisée via un objet fichier, au même titre que celui retourné par l'appel à la fonction *open* lors de l'ouverture d'un fichier. Ces deux descripteurs de fichier implémentent la même méthode *write*, appelée lors de l'utilisation du mot-clé *print*. Dans ce cas-ci, le message n'est pas sauvegardé tout de suite après l'utilisation du mot-clé *print*. Un descripteur de fichier possède habituellement une mémoire tampon pour ne pas sauvegarder immédiatement chaque octet reçu, car cette sauvegarde pourrait causer des ralentissements lorsque, par exemple, l'écriture doit faire appel au disque dur. Un appel à la méthode *flush* de l'objet fichier est alors nécessaire pour forcer la sauvegarde sur disque.

Un comportement plus complexe peut être mis en place avec la création d'une classe personnalisée, qui définit la méthode *write*, appelée lors de l'utilisation de *print*. En lien avec l'exemple précédent, cette classe peut alors s'occuper d'ouvrir le fichier journal lors de la création de l'objet et de faire automatiquement les appels à la méthode *flush*, pour sauvegarder immédiatement le message. Voici un exemple plus complet :

```
import sys

class Imprimeur(object):
```

```
    def __init__(self, nom_fichier):
        self.f = open(nom_fichier, 'w')

    def __del__(self):
        self.f.close()

    def write(self, message):
        self.f.write(message)
        self.f.flush()

sys.stdout = Imprimeur('journal.txt')
```

Si nous avons besoin de retrouver le comportement standard de Python, les variables immuables *sys.__stdin__*, *sys.__stdout__* et *sys.__stderr__* peuvent être utilisées pour retrouver les entrées et sorties standard. Un appel à la méthode *write* de la classe personnalisée pourrait, par exemple, ajouter un préfixe au message à imprimer et effectuer un appel à *sys.__stdout__.write* pour l'afficher à l'écran.

3.8.2 Microprocessus et redirection des entrées/sorties

Dans le module *dstackless*, le concept de classe personnalisée, associée à la variable *sys.stdout*, est utilisé pour intercepter les utilisations du mot-clé *print*. Avant qu'un message soit affiché, cette classe vérifie tout d'abord la configuration du microprocessus qui a fait la demande d'affichage. Son étiquette est retrouvée, ainsi que la configuration de redirection de la sortie standard. Si une étiquette est trouvée, elle est préfixée au message à afficher. Si une redirection des sorties standard est trouvée, le message est envoyé au nœud dont l'identifiant unique se trouve dans cette configuration et c'est celui-ci qui s'occupe de l'afficher. Sinon, le message est tout simplement affiché dans la le terminal associé au nœud où est exécuté le microprocessus. Voici comment spécifier une redirection de la sortie standard d'un microprocessus vers un autre nœud, lors de sa création :

```
def ma_methode(canal):
    # Ce message est affiché sur le
    # noeud identifié par
    # ("127.0.0.1", 6473)
    print "Bonjour"

ma_config_io = {"stdout": ("127.0.0.1", 6473)}
```

```
t = dstackless.tasklet(
    ma_methode,
    ioconfig=ma_config_io
)(mon_canal)
```

La redirection vers un fichier, local ou distant, est tout aussi simple. Il suffit d'utiliser directement l'objet fichier, associé au fichier ouvert, pour la configuration de redirection de la sortie standard. Comme un objet fichier implémente la méthode *write*, cette dernière peut être appelée directement par notre mécanisme, au lieu d'un appel à la méthode *write* de *sys.__stdout__*.

Avant de pouvoir être utilisé, un objet fichier doit être transformé en objet distribué. Cette transformation permet au microprocessus de pouvoir être migré, car une référence directe vers un fichier n'est pas sérialisable. Avec un accès sous forme d'objet distribué, l'objet fichier reste sur le noeud original et lorsque le microprocessus est migré, il utilise alors une référence à distance vers celui-ci.

Pour le cas de l'entrée standard, la situation est un peu plus complexe. La fonction *raw_input*, utilisée pour demander à l'utilisateur d'entrer une valeur, prend comme paramètre un message à afficher. Ce message sert à indiquer à l'utilisateur qu'il doit entrer quelque chose au clavier. L'utilisation de *raw_input* se découpe donc en deux parties, soit l'affichage du message et l'invite de saisie. L'affichage du message se fait bien évidemment à l'aide de la sortie standard.

Les règles de redirection de la sortie standard, définies par le programmeur pour le microprocessus qui utilise *raw_input*, seront donc prises en compte de façon transparente. Pour ce qui est de la saisie de la valeur au clavier, lorsque l'entrée standard est redirigée vers une entrée distante, le microprocessus doit être bloqué en attendant la réponse. Le tout implique donc trois échanges de messages entre les noeuds impliqués : l'affichage de la question, l'envoi de la demande de saisie et finalement le retour de la valeur entrée au clavier.

Étant donné que la redirection de la sortie standard entre en jeu pour afficher la question, le programmeur doit porter attention à bien rediriger l'entrée standard et la sortie standard vers le même nœud. Dans le cas contraire, la question ne serait pas posée sur le nœud où devrait être entrée la réponse.

3.9 Gestion des exceptions

Pour la gestion des exceptions, il n'y a que deux cas à gérer pour couvrir l'ensemble des causes d'arrêt d'un microprocessus. Lorsqu'un microprocessus termine et ne génère aucune exception, le parent ne sera pas informé de cet arrêt, car il ne s'agit normalement pas d'un arrêt causé par un problème. Les deux cas qui nous intéressent touchent la levée d'une exception dans un microprocessus enfant et l'arrêt du nœud exécutant un enfant lorsque le parent se trouve sur un nœud différent.

3.9.1 Relation parent/enfant

Dans Stackless Python, la relation parent/enfant entre les microprocessus n'existe pas. Il est tout de même possible d'utiliser certains mécanismes pour savoir si une exception a été levée à l'intérieur d'un microprocessus. Par exemple, le système de références faibles permet de spécifier une fonction qui est appelée lorsque l'objet associé à une telle référence est nettoyé. Cette fonction est donc appelée lorsqu'un microprocessus termine, peut importe la raison. S'il faut connaître la raison de l'arrêt d'un microprocessus, il faut entourer d'un bloc *try/except/finally* le code de la fonction utilisée pour le créer, pour intercepter l'exception lancée.

Cette façon de faire est tout de même fastidieuse pour le programmeur et ne fonctionne pas bien avec un environnement distribué. Il est difficile de gérer le cas de la migration de microprocessus, ainsi que celui où deux microprocessus en relation sont exécutés sur deux interpréteurs différents et que l'un d'entre eux est arrêté. Gérer ces cas demande des outils pour communiquer directement avec les nœuds, en dehors du monde des microprocessus. On veut par contre éviter d'imposer au programmeur de nouvelles interfaces qui changeraient la façon normale dont les exceptions sont gérées.

Relation parent/enfant des coroutines de PyPy

Dans le module *stackless* de PyPy, une relation parent/enfant existe avec les coroutines, sur lesquelles le module *stackless* se base pour les tasklets. La création d'une coroutine associe automatiquement celle en cours d'exécution en tant que parent de celle nouvellement créée. Toute exception lancée à partir de l'enfant est transmise au parent si elle n'est pas attrapée, à l'exception de *CoroutineExit* qui est lancée pour terminer une coroutine lors de l'appel à *coroutine.kill*.

Pour le parent, si une exception est lancée par un de ses enfants, elle peut être attrapée suite à un appel à *ref_coroutine.switch* ayant permis de donner la main aux autres coroutines. Tout comme les tasklets, les coroutines fonctionnent en concurrence. Il n'y a donc jamais deux coroutines qui sont en état d'exécution en même temps. Lorsqu'une exception est lancée par un enfant, le parent avait donc nécessairement effectué un appel à *ref_coroutine.switch*. Dans la coroutine parente, l'exception est alors levée à l'intérieur de cet appel à *switch*.

La classe *tasklet* du module *stackless* met par contre en place certains mécanismes permettant d'ignorer cette relation et ainsi se comporter de la même manière que la classe *tasklet* de l'implémentation originale de Stackless Python. Il est tout de même possible de se baser sur le comportement des coroutines, et même en utiliser une partie, pour mettre en place une relation parent/enfant entre les microprocessus.

Relation entre microprocessus

La mise en relation de deux objets *tasklet* est offerte de façon optionnelle au programmeur pour rester dans le même esprit que le comportement de base de Stackless Python. Un paramètre nommé *parent* a été ajouté au constructeur de la classe *tasklet*. Il est utilisé pour passer une référence vers l'objet *tasklet* d'un autre microprocessus, qui en deviendra le parent. Cette référence est notée dans l'objet *tasklet* de l'enfant nouvellement créé pour qu'elle soit utilisée par la suite lorsqu'une exception est levée dans celui-ci. Voici un exemple démontrant l'assignation d'un parent lors de la création d'un microprocessus enfant :

```
c = dstackless.channel()
# Parent.
ref_tache_envoyeur = dstackless.tasklet(
        fonction_tache_envoyeur
    )(c)

# Enfant. La référence vers le parent est passée
# au paramètre "parent" du constructeur.
ref_tache_receveur = dstackless.tasklet(
        fonction_tache_receveur,
        parent = ref_tache_envoyeur
    )(c)
```

3.9.2 Injection d'exception

Pour qu'une exception levée par un microprocessus enfant soit transmise au parent, elle doit pouvoir être injectée dans l'objet *tasklet* de celui-ci. Pour répondre à ce besoin, l'implémentation des coroutines de PyPy a dû être modifiée. Cette modification a été simple à mettre en place, car le comportement requis ressemble fortement à celui de la méthode *coroutine.kill*. En effet, cette méthode injecte une exception de type *CoroutineExit* à l'intérieur de la coroutine à terminer.

L'injection d'exception a été ajoutée sous la forme d'une nouvelle méthode nommée *throw*, sur les coroutines. Cette méthode reçoit en argument l'exception à injecter. Étant donné que la classe *tasklet* hérite de la classe *coroutine*, il est donc possible d'utiliser directement cette nouvelle méthode pour la mise en place de la relation parent/enfant. De plus, cet ajout a permis de corriger une lacune dans l'implémentation des fonctionnalités de Stackless Python dans PyPy, par rapport à l'exception *TaskletExit* qui n'était pas lancée lorsque la méthode *kill* d'un objet *tasklet* était appelée.

3.9.3 Gestion des exceptions

Une relation parent/enfant, lorsque créée, est donc utilisée pour gérer les exceptions qui ne sont pas attrapées par un microprocessus enfant. À la création d'un microprocessus, le mécanisme qui s'occupe d'appeler la fonction utilisée comme point d'entrée entoure cet appel d'un bloc *try/except*. Ce bloc intercepte toutes les exceptions levées à l'intérieur d'un microprocessus, pour éviter qu'elles remontent au point où l'interpréteur serait arrêté.

À l'intérieur d'un microprocessus enfant, une exception peut être levée de deux façons. Soit une erreur est survenue et un appel au mot-clé *raise* a été fait pour la signaler, soit l'exception a été injectée dans le microprocessus à l'aide de la méthode *throw* de son objet *tasklet*. Cette exception peut être gérée au niveau de l'enfant ou, sinon, remonter jusqu'à l'appel de son point d'entrée. Lorsqu'elle remonte à ce point, elle est nécessairement attrapée et l'attribut *parent* de l'objet *tasklet* est vérifié.

Lorsque l'attribut *parent* de l'objet *tasklet* d'un microprocessus est défini, donc qu'une relation existe celui-ci et un autre microprocessus, un appel est alors effectué à la méthode *throw* du parent pour lui transmettre l'exception ayant causé l'arrêt de l'enfant. Cet appel peut être fait à distance, à l'aide de l'objet référence du microprocessus parent. Lorsque le microprocessus parent est ordonnancé à nouveau, l'exception est alors levée,

en provenance de son l'appel à *stackless.schedule*. Une telle exception peut aussi être levée lors d'un appel bloquant à *channel.send* ou *channel.receive*, car ces derniers font aussi un appel à *stackless.schedule* pour bloquer le microprocessus qui effectue la requête sur le canal.

L'exception provenant de l'enfant n'est pas envoyée telle quelle au parent. Elle est tout d'abord entourée dans une exception de type *ChildException*, avec une référence vers l'enfant qui l'a levée. C'est cette nouvelle exception est envoyée au parent qui peut alors savoir quel enfant s'est arrêté et quelle en est la cause. Le prochain exemple démontre comment est faite cette gestion au niveau du module *dstackless*. Ce code remplace celui de la méthode *taskless.setup* du module *stackless* de PyPy, qui ignore les exceptions.

```
try:
    # Appel du point d'entrée du microprocessus.
    func(*argl, **argd)
except Exception, e:
    if self.parent is not None:
        self.parent.throw(ChildException(e, self))
finally:
    # Fin de l'exécution du microprocessus.
    stackless._scheduler_remove(self)
    self.alive = False
```

L'arrêt du nœud où est exécuté un microprocessus enfant doit être traité d'une autre manière lorsque le parent ne se trouve pas sur le même nœud. Pour répondre à ce besoin, lorsque le microprocessus enfant est créé et qu'un parent lui est attribué, le parent doit lui aussi se voir attribuer une référence vers son enfant. Cette référence est ajoutée dans une liste d'enfants, car il peut en avoir plusieurs. Le nœud à l'intérieur duquel se trouve le parent a alors un rôle important lors de la disparition du nœud qui exécute l'enfant.

Comme c'est au niveau du nœud que la déconnexion d'un autre nœud peut être détectée, celui-ci doit alors parcourir tous les microprocessus qu'il exécute pour vérifier s'il n'y a pas un d'entre eux qui a un enfant qui s'exécutait sur le nœud disparu. Lorsqu'il trouve un tel microprocessus, il doit alors lui injecter une exception, pour lui indiquer que son enfant n'existe plus. Encore une fois, cette exception est de type *ChildException*. La référence vers l'enfant disparu et une exception de type *NodeHalted* sont jointes à celle-ci.

3.10 Interface de programmation

L'implémentation des concepts liés à la migration de microprocessus est offerte au programmeur sous forme d'une interface de programmation ressemblant fortement à celle de Stackless Python. Cette interface fait par contre partie d'un nouveau module nommé *dstackless*. Les fonctions de Stackless Python sont toujours disponibles, mais certaines ont été ajoutées pour la création de nœuds, la migration de microprocessus ainsi que la création d'objets distribués. Cette section explique chacune des fonctions disponibles dans le module *dstackless*.

3.10.1 Fonctions et méthodes provenant de Stackless Python

dstackless.channel()

Créer un canal de communication bloquant. Ce canal peut échanger des objets de façon locale à un nœud, mais contrairement aux canaux de Stackless Python, il peut être utilisé pour l'échange d'objets entre deux nœuds.

Valeur de retour :
- Objet canal nouvellement créé.

Exemple :

```
c = dstackless.channel()
t = dstackless.tasklet(fonction)(c)
```

channel.send(data)

Envoi bloquant d'un objet sur un canal. Tant et aussi longtemps qu'une demande de réception n'est pas effectuée sur le canal, le microprocessus envoyeur reste bloqué. Lorsque plusieurs demandes d'envoi sont en attente, elles sont servies dans l'ordre d'arrivée.

Paramètres :
- *data* : Message à envoyer sur le canal ; doit être sérialisable.

Exemple :

```
def envoyeur(objet_canal):
    objet_canal.send("Bonjour monde!")
```

channel.receive()

Réception bloquante d'un message sur un canal. Le receveur reste bloqué tant et aussi longtemps qu'il n'y a pas de requête d'envoi sur le canal. Lorsque plusieurs demandes de réception sont en attente, elles sont servies dans l'ordre d'arrivée.

Valeur de retour :
- Objet ayant été reçu par le canal de communication.

Exemple :

```
def receveur(objet_canal):
    message = objet_canal.receive()
    print message
```

dstackless.tasklet()

Création d'un microprocessus. Une fois l'objet *tasklet* créé, il doit être appelé comme une fonction pour démarrer le microprocessus. Les paramètres d'appel de l'objet sont ceux qui seront envoyés à la fonction utilisée comme point d'entrée du microprocessus. Dès que cet appel est effectué, l'objet *tasklet* est immédiatement inséré dans la file d'attente de l'ordonnanceur.

Paramètres :
- *func* : Fonction Python utilisée comme point d'entrée du microprocessus.
- *label (Optionnel – nouveau avec dstackless)* : Étiquette permettant d'identifier le microprocessus. Cette étiquette préfixe les messages affichés lors des appels au mot-clé *print* par le microprocessus.
- *ioconfig (Optionnel – nouveau avec dstackless)* : Configuration des entrées/sorties du microprocessus. Accepte un dictionnaire contenant l'identifiant du nœud de destination de la sortie standard identifiée par la clé *stdout*. Voici un exemple : {"stdout" : (*localhost*, 6474)}

– *parent (Optionnel – nouveau avec dstackless)* : Référence vers un objet *tasklet* qui sera considéré comme le parent du nouveau microprocessus. Toutes les exceptions levées dans le microprocessus et qui ne sont pas attrapées sont envoyées au parent.

Valeur de retour :

– Objet *tasklet* associé au microprocessus.

Exemple :

```
def receveur(objet_canal):
    message = objet_canal.receive()
    print message

def envoyeur(objet_canal):
    print 'Message affiché dans le terminal', \
        'du nœud ("localhost", 6473)'
    objet_canal.send("Bonjour monde!")

c = dstackless.channel()
ref_envoyeur = dstackless.tasklet(
    envoyeur,
    ioconfig={
        "stdout": ("localhost", 6474)
    }
)

# "c" est passé à la fonction "envoyeur".
ref_envoyeur(c)

# L'appel de l'objet "tasklet" peut être
# effectué directement lors de sa création.
ref_receveur = dstackless.tasklet(
    receveur,
    parent=ref_envoyeur
)(c)

# Lancement de l'ordonnanceur.
dstackless.run()
```

dstackless.run()

Démarrage de l'ordonnanceur de Stackless Python. C'est à partir de ce moment que les microprocessus, présents dans le nœud courant, sont exécutés en concurrence.

ref_tasklet.kill()

Arrêter l'exécution d'un microprocessus. Le microprocessus peut être en cours d'exécution sur le nœud où l'appel est effectué ou il peut se trouver sur un nœud distant. Une exception de type *TaskletExit* est levée à même le microprocessus, qui peut l'intercepter.

3.10.2 Nouvelles fonctions et méthodes

dstackless.identify()

Identifier de façon unique un nœud à l'intérieur du réseau de nœuds. L'interpréteur écoute sur le port *TCP* faisant partie de l'identifiant unique pour recevoir des requêtes en provenance de nœuds distants.

Paramètres :
- *Identifiant* : N-uplet (IP-Hôte, Port)

Exemple :

```
dstackless.identify(("localhost", 6473))
```

ref_tasklet.move()

Migration d'un microprocessus en cours d'exécution vers le nœud identifié par le n-uplet (IP-Hôte, Port).

Paramètres :
- *identifier* : Identifiant unique du nœud de destination sous forme d'un n-uplet (IP-Hôte, Port).

Valeur de retour :
- *True* lorsque la migration a été effectuée avec succès.
- *False* lorsque la tentative de migration s'est terminée avec une erreur.

Exemple :

```
ref_tasklet.move(("localhost", 6474))
```

dstackless.ref_object()

Création d'un objet distribué. Un tel objet n'est pas copié lorsqu'il est référé par un microprocessus migré ou qu'il est envoyé via un canal de communication. Seule une référence à distance est créée vers celui-ci.

Paramètres :
- *obj* : Objet à transformer sous forme d'objet distribué.

Valeur de retour :
- Nouvel objet référence associé à l'objet passé en paramètre.

Exemple :

```
mon_objet = NouvelObjet()
mon_objet_distribue = dstackless.ref_object(mon_objet)
objet_canal.send(mon_objet_distribue)
```

dstackless.sleep()

Interrompre l'exécution d'un microprocessus pendant un certain temps sans bloquer les autres microprocessus en cours d'exécution.

Paramètres :
- *delay* : Délai d'attente en secondes, sous forme de nombre à virgule flottante).

Exemple :

```
dstackless.sleep(1.5)
```

3.11 Conclusion

Dans ce chapitre, nous avons donc vu la conception de *dstackless*, un module basé sur des microprocessus communiquant par canaux de communication et pouvant être migrés d'un système à un autre de façon transparente. Offrir une telle possibilité de migration a par contre requis un environnement homogène : il faut pouvoir sauvegarder l'état d'exécution d'un microprocessus, ainsi que son environnement mémoire, dans un format compatible avec tous les systèmes qu'il visitera. Compte tenu de la diversité des architectures matérielles et des systèmes d'exploitation, il a été convenu que seul un langage interprété peut offrir un tel environnement homogène.

Pour répondre au besoin d'utiliser un langage interprété, PyPy a été choisi comme point de départ. Cette implémentation alternative du langage Python implémente les fonctionnalités de Stackless Python et ajoute ainsi un concept de microprocessus, les *tasklets*, qui communiquent par canaux. Ces *tasklets* sont basés sur les continuations ; la possibilité de sauvegarder l'état d'exécution de ceux-ci est offerte au programmeur. Toutes les bases nécessaires à la création d'une version distribuée de Stackless Python, basée sur la migration de microprocessus, étaient donc présentes.

Nous voulions tout d'abord limiter les ajouts au langage. Ceci a pour but de pouvoir prendre des programmes écrits avec Stackless Python et les convertir, avec peu de modifications, en programmes distribués. Les canaux de communication ont donc été transformés en canaux hybrides pouvant effectuer des communications locales à l'intérieur d'un seul interpréteur ou à distance avec un autre interpréteur. Un message envoyé sur un canal peut donc être reçu par un autre microprocessus exécuté sur le même interpréteur, ou par un microprocessus présent sur un autre interpréteur, et le tout est géré de façon transparente pour le programmeur.

Ensuite, la notion de migration a été introduite sur les microprocessus à l'aide d'une méthode nommée *move*. Cette dernière permet de migrer immédiatement un microprocessus en cours d'exécution vers un interpréteur distant. Le processus de migration s'occupe de tout ce qui est nécessaire pour que la récupération de l'état d'exécution soit possible. L'état d'exécution, l'environnement mémoire ainsi que le code sont transférés de façon automatique. De plus, tous les liens entre microprocessus, générés par le partage de canaux, sont maintenus.

Certains problèmes on tout de même dû être résolus. Malgré que la communication entre deux microprocessus soit basée sur les canaux, un partage d'objets demeure possible. Dans un langage objet, lorsqu'un objet est passé en paramètre à une fonc-

tion, seule une référence vers celui-ci est passée. Le même phénomène survient lors des échanges via les canaux de Stackless Python. Ce partage a une conséquence pour les migrations de microprocessus, ainsi que pour les communications via un canal entre deux interpréteurs, car une copie des objets devient obligatoire. En effet, l'échange entre deux interpréteurs doit passer par les mêmes méthodes de communication interprocessus que les systèmes d'exploitation et qui fonctionnent nécessairement par copie d'un flot d'octets.

Malgré la copie imposée par le passage sur le réseau, un outil a été mis en place pour la création d'objets distribués pour lesquels seule une référence est échangée. De tels objets résident donc sur un seul nœud et toute opération effectuée à partir de la référence est redirigée vers ce dernier. La possibilité de créer des objets distribués règle aussi, en partie, les problèmes liés aux objets d'entrées/sorties tels que les fichiers et les connexions réseau qui ne peuvent être copiés. Pour mettre en place ces objets distribués, le système d'objets mandataires, spécifique a PyPy a été d'une grande aide.

Le cas de l'entrée standard et de la sortie standard est un peu différent. Ces derniers sont des objets présents de façon implicite au démarrage d'un programme et sont normalement liés au terminal dans lequel le programme s'exécute. La plupart des langages, incluant Python, offrent la possibilité de les intercepter pour mettre en place une gestion personnalisée. Ce mécanisme a donc été utilisé pour offrir au programmeur de les rediriger, et ce, pour chacun des microprocessus. Par exemple, la sortie standard peut être redirigée vers un fichier journal ou vers un interpréteur en particulier, et ce, même si le microprocessus est migré.

Finalement, une relation parent-enfant entre microprocessus a été introduite, ainsi qu'une redirection des exceptions pour que le parent soit informé de la terminaison d'un de ses enfants. Ce mécanisme devient nécessaire dans un environnement distribué pour éviter que des microprocessus demeurent indéfiniment en attente d'un message et polluent ainsi l'espace mémoire d'un interpréteur. Une méthode permettant d'injecter une exception à l'intérieur d'un objet *tasklet* a dû être mise en place pour répondre à ce besoin.

Le résultat de cette implémentation est le module *dstackless*, qui offre une interface de programmation très simple. Des programmes existants peuvent être transformés en programmes distribués avec peu de modifications. Le prochain chapitre présente quelques exemples de l'utilisation de ce nouveau module.

Chapitre 4

Applications

Maintenant que nous avons une implémentation des outils permettant la migration de microprocessus sous forme du module *dstackless*, nous sommes en mesure de créer des programmes utilisant ces outils. Ce chapitre présente deux types d'applications distribuées. Ces applications sont tout d'abord expliquées et une implémentation est par la suite présentée à l'aide de *dstackless*. À l'aide de ces exemples, une méthode permettant de modifier le modèle distribué d'un programme ainsi que les modifications nécessaires pour transformer un programme Stackless Python en programme distribué sont aussi énoncées.

4.1 Agents mobiles

Un agent mobile est un programme informatique capable de se déplacer d'un système à un autre et de continuer son exécution sur chacun des systèmes qu'il visite de façon autonome. Il est capable d'effectuer des opérations et de récolter de l'information sur chacun de ces systèmes et peut prendre des décisions selon ce qu'il a récolté. Lors de ses déplacements, un agent emporte avec lui toute cette information, ainsi que son code, ce qui évite la nécessité de le déployer au préalable. À ce propos, la littérature mentionne régulièrement les travaux effectués par *General Magic* qui décrivent les conditions nécessaires pour la mise en place d'un système basé sur les agents mobiles [49] [30]. Ils ont aussi été comparés à des technologies alternatives comme l'appel de procédure à distance (*Remote Procedure Call*) ou l'envoi et l'exécution automatique de programmes sur des systèmes à distance [9].

À la différence des agents mobiles, l'appel de procédure à distance offre la possibilité d'appeler à distance des fonctions qu'un système distant rend disponibles. Lors de l'appel de la fonction, une requête contenant le nom de celle-ci ainsi que les paramètres d'appels est envoyée au système qui l'héberge. La fonction est alors exécutée sur ce système et la valeur de retour est envoyée au système ayant effectué l'appel.

Avec cette structure, il est donc possible d'exécuter différentes parties d'un programme sur différents systèmes et même d'effectuer certains traitements en parallèle, car une même procédure peut être déployée sur plusieurs systèmes. Il est alors possible de découper un ensemble de données à traiter en de plus petits sous-ensembles et d'appeler en parallèle les différentes instances de la procédure dans le but d'accélérer le traitement. Cette façon de faire à tout de même un inconvénient majeur. Pour rendre les procédures disponibles, leur code doit au préalable être déployé sur chacun des systèmes qui seront utilisés.

De son côté, l'envoi et l'exécution automatique de programmes à distance éliminent la nécessité de déployer manuellement le code. Cette façon de faire se présente habituellement sous la forme d'un outil qui reçoit, un programme à appliquer à chacun des éléments d'un lot de données à traiter, le lot en question et une liste de systèmes ayant la capacité d'effectuer le traitement. L'exécution du programme distribuée est alors divisée en trois étapes :

1. La première étape est d'envoyer une copie du programme et une partie du lot à traiter à chacun des systèmes. Pour ce faire, le protocole *SSH*, disponible sur la majorité des systèmes UNIX, peut être utilisé.
2. Par la suite, une session *SSH* est automatiquement ouverte sur chacun des systèmes et le programme envoyé est exécuté.
3. Une fois l'exécution terminée, le résultat est retrouvé sur le système à partir duquel l'exécution a été démarrée.

Certains outils utilisent des protocoles d'envoi et de démarrage qui leur sont propres et peuvent demander le déploiement préalable d'un serveur générique pouvant exécuter tout ce qui lui est envoyé. *Parallel Python* [20] est un bon exemple de système faisant partie de cette catégorie.

Pour la mise en place des agents mobiles, certaines conditions sont nécessaires. La première est le besoin de disposer d'un moyen de communication entre les systèmes qu'un agent a à visiter. Il doit être en mesure de communiquer avec chacun d'entre eux pour pouvoir se déplacer. Ce moyen de communication peut aussi être utilisé pour envoyer des messages au système à partir duquel un agent a été initialisé. Par exemple, des messages pourraient être envoyés pour indiquer la progression de son travail.

On retrouve ensuite la persistance. Un agent doit être en mesure de sauvegarder son état d'exécution et de le récupérer par la suite. Cette sauvegarde doit se faire dans un format pouvant être transféré via un réseau. Lorsque l'agent est transféré, il doit aussi emporter avec lui tout le code nécessaire à la poursuite de son exécution sur le nouveau système. C'est cette façon de procéder qui élimine la nécessité de déployer du code sur chacun des systèmes à visiter.

Finalement, une homogénéité doit exister pour l'ensemble des systèmes qu'un agent doit visiter. Le contexte et l'état d'exécution d'un agent doivent être sauvegardés dans un format compatible avec tous ces systèmes. Si un agent est programmé dans un langage compilé, son utilisation se limite donc aux systèmes basés sur la même architecture matérielle et logicielle. En réalité, une telle contrainte limite la possibilité de faire évoluer les différents systèmes qui seront utilisés. L'utilisation d'un langage interprété évite ce problème, car les programmes n'ont pas à être modifiés pour pouvoir être exécutés sur différentes architectures. L'interpréteur peut alors être adapté à chacune d'entre elles et masquer leurs différences aux programmes.

4.1.1 Exemple dstackless

L'utilisation de microprocessus pouvant être migrés remplis donc toutes les conditions nécessaires pour la création d'agents mobiles. Sur chacun des systèmes que l'agent doit visiter, un interpréteur, disposant du module *dstackless*, est démarré pour les transformer en nœuds. Un microprocessus peut par la suite se migrer sur chacun de ces nœuds et effectuer les opérations qu'il a à faire. Prenons en exemple un système de déploiement automatisé d'applications basé sur les agents mobiles. Un microprocessus moniteur et un microprocessus agent sont démarrés. Le moniteur s'occupe d'afficher l'information que l'agent lui retourne et l'agent saute de nœud en nœud pour déployer son application.

```
import dstackless
import status

def agent(canal_statut, paquet_application, liste_noeud):
    for noeud in liste_noeud:
        dstackless.get_current().move(noeud)

        try:
            deployer_paquet(paquet_application)
            canal_statut.send((noeud, status.DONE))
        except ErreurDeploiement:
```

```
            canal_statut.send((noeud, status.FAIL))

def moniteur(status_channel):
    while True:
        try:
            noeud, code_statut = canal_statut.receive()
        except ChildException, e:
            print "L'exécution de l'agent s'est", \
                "terminée en erreur..."
            return

        if code_statut == status.DONE:
            print "Déploiement effectué avec succès sur", \
                noeud
        elif code_statut == status.FAIL::
            print "Erreur de déploiement sur", noeud

if __name__ == '__main__':
    dstackless.identify(("localhost", 6473))

    canal_statut = dstackless.channel()

    paquet_application = creer_paquet()

    liste_noeud = (
        ("192.168.1.1", 6473),
        ("192.168.1.2", 6474),
        ("192.168.1.3", 6475)
    )

    ref_moniteur = dstackless.tasklet(
        moniteur
    )(canal_statut)

    dstackless.tasklet(
        agent,
        parent=ref_moniteur
    )(
        canal_statut,
        paquet_application,
```

```
        liste_noeud
    )

    dstackless.run()
```

Un canal est tout d'abord créé pour que l'agent puisse envoyer le résultat des opérations au moniteur. Le moniteur est par la suite démarré et reçoit ce canal. Il s'occupe alors d'afficher le résultat de chacune des opérations de déploiement de l'agent. Finalement, l'agent est démarré et reçoit le canal de communication sur lequel il devra envoyer son statut, un paquet représentant l'application à déployer et une liste de nœuds sur lesquels il doit déployer l'application. Une fois démarré, il se déplace sur chacun des nœuds, installe l'application et retourne le résultat sur le canal. Si le déploiement échoue sur un nœud, l'agent continue quand même son exécution sur les autres nœuds, après avoir informé le microprocessus moniteur de l'échec du déploiement. Une relation parent/enfant est créée entre l'agent et le moniteur pour aviser ce dernier si l'agent se termine de façon inopinée. Un message est alors affiché.

Un point intéressant avec cette structure est que peu de modifications sont nécessaires pour transformer ce programme composé d'un seul agent mobile en un programme comportant plusieurs agents distribués pouvant effectuer les installations de façon parallèle. Pour ce, il suffit de démarrer plusieurs microprocessus *agent* en parallèle en leurs envoyant une liste composée d'un seul nœud, au lieu d'envoyer la liste complète des nœuds à un seul agent. Voici le code modifié.

```
if __name__ == '__main__':
    dstackless.identify(("localhost", 6473))

    canal_statut = dstackless.channel()

    paquet_application = creer_paquet()

    liste_noeud = (
        ("192.168.1.1", 6473),
        ("192.168.1.2", 6474),
        ("192.168.1.3", 6475)
    )

     ref_moniteur = dstackless.tasklet(
         moniteur
     )(canal_statut)
```

```
#Cette boucle est ajoutée pour démarrer
#plusieurs agents en parallèle qui reçoivent chacun
#un seul noeud.
for noeud in liste_noeud:
    dstackless.tasklet(
        agent,
        parent=ref_moniteur
    )(
        canal_statut,
        paquet_application,
        [noeud]
    )

dstackless.run()
```

4.2 MapReduce

Un bon exemple qui se prête bien à l'utilisation de microprocessus est l'outil *MapReduce* développé par Google [6]. Le besoin derrière cet outil est le traitement d'un très grand volume de données dans un délai raisonnable et de façon distribuée. L'idée est simple et s'inspire des fonctions *map* et *reduce* provenant du monde des langages fonctionnels. Avec ces langages, la fonction *map* est utilisée pour appliquer une fonction à chacun des éléments d'une liste. Elle prend donc en paramètre une fonction ainsi qu'une liste d'éléments à traiter. Elle retourne ensuite une liste qui contient les résultats de l'application de la fonction à chacun des éléments de la liste reçue. Voici un exemple, qui consiste à ajouter 5 à chacun des nombres présents dans une liste :

```
>>> listeSortie = map(lambda x: x + 5, [1, 2, 3, 4, 5])
>>> listeSortie
[6, 7, 8, 9, 10]
```

De son côté, la fonction *reduce* prend aussi une fonction et une liste en entrée, mais son rôle est différent. La fonction donnée en entrée doit recevoir deux paramètres. Elle s'occupe de rassembler les éléments de la liste pour générer un résultat unique. Les deux premiers éléments de la liste sont tout d'abord réduits avec la fonction ; le résultat obtenu et le prochain élément de la liste sont réduits ; et ainsi de suite. Cette méthode est appliquée tant et aussi longtemps qu'il y a des éléments dans la liste. Pour continuer l'exemple précédent, voici une utilisation de *reduce* permettant d'additionner

les éléments de la liste obtenue suite à l'utilisation de la fonction *map* :

```
>>> somme = reduce(lambda x, y: x + y, listeSortie)
>>> somme
40
```

La valeur 40 est obtenue à l'aide de la suite d'additions suivante :

```
6 + 7 = 13
13 + 8 = 21
21 + 9 = 30
30 + 10 = 40
```

4.2.1 Utilisation distribuée

L'outil *MapReduce* de Google a donc pour but d'amener les fonctions *map* et *reduce* dans un monde distribué. Ce but est simple à atteindre, car le traitement effectué par ces fonctions peut facilement être mis en parallèle. Prenons tout d'abord les définitions utilisées par cet outil pour les fonctions *map* et *reduce*.

```
map (k1,v1) -> list(k2,v2)
reduce (k2,list(v2)) -> list(v2)
```

La fonction *map* prend en entrée une clé *k1* représentant un ensemble de données à traiter et une liste *v1* qui contient un sous-ensemble de ces données. En retour, des éléments *v2* identifiés par des clés *k2* et faisant partie d'un autre domaine de données sont émis. Ces nouveaux éléments sont rassemblés par la fonction *reduce*. Pour chaque clé *k2* unique, les listes d'éléments *v2* obtenues par différents appels à *map* sont rassemblées et une unique liste est retournée. Voici un exemple simple tiré de [6], qui consiste à compter le nombre d'occurrences des différents mots d'un documents :

```
map(String key, String value):
    // key: Nom du document
    // value: Contenu du document
    for each word w in value:
        EmitIntermediate(w, "1");

reduce(String key, Iterator values):
    // key: Un mot
    // values: Une liste de décomptes du mot
```

```
int result = 0;
for each v in values:
    result += ParseInt(v);
Emit(AsString(result));
```

La fonction *map* reçoit le contenu d'un document et pour chacun des mots trouvés, une valeur intermédiaire est émise. La clé de cette valeur (*k2*) est le mot trouvé et la valeur (*v2*) est le décompte *1* (une occurrence du mot trouvée). Par la suite, pour chacune des clés *k2* intermédiaires, la fonction *reduce* est appelée avec la liste des décomptes *v2* intermédiaires. Ces décomptes sont additionnés et le résultat est retourné. Ce qui est intéressant dans cet exemple est que la fonction *map* peut être appelée en parallèle pour chacun des documents d'un ensemble de documents à traiter. Chacun de ces appels peut être exécuté sur un système différent et les multiples résultats intermédiaires obtenus sont stockés dans un endroit commun. Finalement, des appels parallèles à *reduce* peuvent être effectués pour chacun des documents en entrée et un dernier appel à *reduce* peut rassembler les résultats obtenus pour l'ensemble des documents.

4.2.2 Version dstackless

L'exemple du comptage des occurrences des différents mots d'un document peut facilement être reproduit avec Stackless Python. Un microprocessus est utilisé pour la lecture du document. Plusieurs microprocessus concurrents s'occupent de représenter la fonction *map*, qui compte les différents mots. Finalement, un microprocessus représente la fonction *reduce*, qui rassemble les résultats intermédiaires émis par chacun des microprocessus *map*. Voici le code d'un tel programme :

```
import stackless
import types

def map(canal_lignes, canal_resultat):
    while True:
        resultat = {}

        liste_lignes = canal_lignes.receive()

        for ligne in liste_lignes:
            liste_mots = ligne.split()
            for mot in liste_mots:
                if mot not in resultat:
```

```
                    resultat[mot] = 1
                else:
                    resultat[mot] = resultat[mot] + 1

        canal_resultat.send(resultat)

def lecteur(nom_fichier, canal_lignes, canal_resultat):
    f = open(nom_fichier, "r")
    ligne = f.readline()

    decompte = 0
    bloc_donnees = 0
    liste_lignes = []
    while ligne != "":
        liste_lignes.append(ligne)
        ligne = f.readline()
        decompte = decompte + 1
        if decompte == 10000 or ligne == "":
            canal_lignes.send(liste_lignes)
            bloc_donnees = bloc_donnees + 1
            decompte = 0
            liste_lignes = []

    canal_resultat.send(bloc_donnees)

def reduce(canal_resultat):
    resultat = {}
    bloc_donnees = None
    resultats_recus = 0
    while bloc_donnees is None or \
            resultats_recus != bloc_donnees:
        bloc_resultat = canal_resultat.receive()

        if type(bloc_resultat) is types.IntType:
            bloc_donnees = bloc_resultat
        elif bloc_resultat is not None:
            for mot in bloc_resultat:
                if not mot in resultat:
                    resultat[mot] = bloc_resultat[mot]
                else:
```

```
                    resultat[mot] = resultat[mot] + \
                        bloc_resultat[mot]
                resultats_recus = resultats_recus + 1

        print "Nombre de différents mots:", len(resultat)

if __name__ == '__main__':
    canal_lignes = stackless.channel()
    canal_resultat = stackless.channel()

    stackless.tasklet(lecteur)(
                    "fichier_texte.txt",
                    canal_lignes,
                    canal_resultat
                )
    stackless.tasklet(map)(canal_lignes, canal_resultat)
    stackless.tasklet(map)(canal_lignes, canal_resultat)
    stackless.tasklet(reduce)(canal_resultat)

    stackless.run()
```

Dans cet exemple, le microprocessus *lecteur* partage un canal de communication *canal_lignes* avec les différents microprocessus *map*. À chaque fois que *lecteur* lit 10000 lignes du fichier d'entrée, il envoie l'ensemble lu sur le canal *canal_lignes*. Le premier microprocessus *map* à lire sur ce canal reçoit l'ensemble de données et le traite. Une fois traité, le résultat est envoyé sur le canal *channel_result* utilisé par le microprocessus *reduce*. À chaque fois que ce dernier reçoit un résultat intermédiaire, il le rassemble dans son dictionnaire qui contiendra le résultat final. Le microprocessus *lecteur* envoie, à la fin de son exécution au microprocessus *reduce*, le nombre d'ensembles de données qu'il a envoyées aux microprocessus *map*. Ce décompte est utilisé par *reduce* pour savoir lorsqu'il doit arrêter de traiter les résultats intermédiaires et afficher le résultat final.

Malheureusement, avec l'implémentation actuelle de Stackless Python, les différents microprocessus *map* ne peuvent être exécutés en parallèle, car ils sont exécutés en concurrence collaborative à l'intérieur d'un seul processus. Même si chacun d'eux pouvait être exécuté dans son propre fil d'exécution, le verrou global de l'interpréteur, que la plupart des interpréteurs du langage Python implémentent, limite l'accès à l'ensemble des variables à un seul fil d'exécution à la fois. Les microprocessus seraient donc de toute façon exécutés un à la suite de l'autre.

Pour obtenir une réelle exécution parallèle des différents microprocessus *map*, le module *dstackless* peut être utilisé. L'idée est d'utiliser plusieurs instances de l'interpréteur pour exécuter chacun des microprocessus *map*. Les problèmes liés au verrou global d'interpréteur sont donc éliminés et un réel parallélisme peut avoir lieu. De plus, peu de modifications sont à faire à l'intérieur du programme original pour supporter le modèle distribué du module *dstackless*. Voici le programme modifié. Des commentaires à même le code indiquent les modifications apportées.

```
# Import de stackless modifié pour dstackless.
import dstackless
import types

def map(canal_lignes, canal_resultat, dest):
    # Ajout d'une opération de migration.
    # La destination est fournie au démarrage du
    # microprocessus.
    dstackless.getcurrent().move(dest)

    while True:
        resultat = {}

        liste_lignes = canal_lignes.receive()

        for ligne in liste_lignes:
            liste_mots = ligne.split()
            for mot in liste_mots:
                if mot not in resultat:
                    resultat[mot] = 1
                else:
                    resultat[mot] = resultat[mot] + 1

        canal_resultat.send(resultat)

def lecteur(nom_fichier, canal_lignes, canal_resultat):
    f = open(nom_fichier, "r")
    ligne = f.readline()

    decompte = 0
    bloc_donnees = 0
    liste_lignes = []
    while ligne != "":
```

```
            liste_lignes.append(ligne)
            ligne = f.readline()
            decompte = decompte + 1
            if decompte == 10000 or ligne == "":
                canal_lignes.send(liste_lignes)
                bloc_donnees = bloc_donnees + 1
                decompte = 0
                liste_lignes = []

        canal_resultat.send(bloc_donnees)

    def reduce(canal_resultat, dest):
        # Ajout d'une opération de migration.
        # La destination est fournie au démarrage du
        # microprocessus.
        dstackless.getcurrent().move(dest)

        resultat = {}
        bloc_donnees = None
        resultats_recus = 0
        while bloc_donnees is None or \
                resultats_recus != bloc_donnees:
            bloc_resultat = canal_resultat.receive()

            if type(bloc_resultat) is types.IntType:
                bloc_donnees = bloc_resultat
            elif bloc_resultat is not None:
                for mot in bloc_resultat:
                    if not mot in resultat:
                        resultat[mot] = bloc_resultat[mot]
                    else:
                        resultat[mot] = resultat[mot] + \
                            bloc_resultat[mot]
                resultats_recus = resultats_recus + 1

        print "Nombre de différents mots:", len(resultat)

    if __name__ == "__main__":
        # Identification du noeud.
        dstackless.identify(("localhost", 6474))
```

```
canal_lignes = dstackless.channel()
canal_resultat = dstackless.channel()

# Tous les appels au module "stackless"
# sont remplacés par des appels au module
# "dstackless". Les fonctions appelées sont les mêmes.
stackless.tasklet(lecteur)(
                "fichier_texte.txt",
                canal_lignes,
                canal_resultat
            )

# Le noeud de destination est envoyé
# à chacun des microprocessus
# devant être migré.
dstackless.tasklet(map)(
                canal_lignes,
                canal_resultat,
                ("localhost", 6473)
            )
dstackless.tasklet(map)(
                canal_lignes,
                canal_resultat,
                ("localhost", 6475)
            )
dstackless.tasklet(reduce)(
                canal_resultat,
                ("localhost", 6476)
            )

dstackless.run()
```

Outre le changement du module *stackless* pour *dstackless*, les seuls changements notables sont l'ajout d'un identifiant unique pour transformer l'interpréteur en nœud, l'ajout d'appels à *tasklet.move* et l'ajout d'un nouveau paramètre *dest* aux fonctions *map* et *reduce*. Ce nouveau paramètre est utilisé pour envoyer aux microprocessus l'identifiant unique des noeuds vers lesquels ils doivent être migrés. Lorsque les microprocessus associés sont démarrés, il se déplacent tout d'abord sur chacun des nœuds reçus en paramètre et ils effectuent ensuite leurs traitements sur ces ceux-ci.

Conclusion

Les travaux ayant rendu possible l'élaboration de ce mémoire ont permis d'explorer les différentes facettes de la programmation parallèle et distribuée. Bien que de multiples modèles permettant ce type de programmation existent depuis quelques dizaines d'années, ce n'est que tout récemment qu'ils ont obtenu une attention particulière de la part de l'industrie. En effet, ce n'est plus l'amélioration de la vitesse des processeurs qui est recherchée pour améliorer les performances d'un système ; les limites physiques des processeurs ont fait que des méthodes alternatives ont dû être explorées.

La solution à la limite physique des processeurs s'étant le plus démarquée est l'ajout de cœurs aux processeurs. Ces multiples cœurs peuvent être vus comme plusieurs processeurs en un. Ils permettent donc un réel parallélisme en étant capables d'exécuter en même temps différents programmes ou plusieurs tâches d'un même programme. Dans un autre ordre d'idée, la recherche de la puissance de traitement s'est aussi faite à l'aide de la connexion de plusieurs ordinateurs entre eux pour former un système distribué. Certains de ces superordinateurs peuvent être composés de centaines d'ordinateurs.

Le but principal de ce travail de maîtrise était donc de trouver un moyen pour simplifier la création de programmes parallèles pouvant exploiter les processeurs composés de multiples cœurs et les systèmes distribués. L'idée était de voir chacun de ces éléments comme des unités de traitement distinctes pouvant être utilisées de la même façon. À cette fin, différentes algèbres de processus et langages de programmation basés sur la concurrence ou le parallélisme ont été étudiés. Une comparaison a permis de voir qu'aucun d'eux ne se démarque des autres. Chacun des concepts explorés est partagé par plusieurs de ces langages et c'est le choix de ces concepts qui différencie chacun des autres.

D'un côté, les algèbres de processus expriment de façon plus abstraite des systèmes concurrents composés de processus pouvant communiquer entre eux. Ils permettent de décrire à très haut niveau les processus composant un programme ainsi que les différentes possibilités de communication entre ces processus. Cette description permet par

la suite de vérifier certaines propriétés comme l'absence d'interblocage. Cette possibilité devient intéressante lorsque la structure d'un programme écrit dans un langage de programmation peut être exprimée à l'aide d'une algèbre de processus. Pour cette raison, les langages de programmation que nous avons étudiés sont tous basés sur des algèbres de processus ou s'en inspirent. Ils mettent en place des systèmes basés sur des processus concurrents devant utiliser des canaux de communication ou un système de boîte aux lettres pour échanger des données. Ces propriétés offrent une plus grande souplesse que la concurrence basée sur les fils d'exécution dans un environnement à mémoire partagée.

Cette étude des algèbres de processus et des langages de programmation parallèles a finalement permis la conception d'un modèle de programmation basé sur la migration de processus. Ce modèle a été créé en lien avec les langages objets qui imposent certaines contraintes qui sont absentes des langages fonctionnels ayant été étudiés. Dans la plupart des cas, il n'existe pas de solution magique permettant de résoudre un problème et des outils et méthodes ont dû être proposés pour donner le choix de la solution appropriée selon le besoin du programmeur. Le modèle de programmation proposé a été implémenté sous la forme d'un module nommé *dstackless*, qui permet d'étendre les fonctionnalités de Stackless Python. Ce module a été développé de façon à limiter les changements nécessaires pour transformer des programmes existants en programmes distribués.

Travaux futurs

L'interface de programmation de *dstackless* offre un ensemble minimal de fonctionnalités permettant la construction d'outils distribués plus complexes. Des outils de surveillance permettant une gestion manuelle de la migration des microprocessus ainsi que des objets distribués pourraient être créés. Ils permettraient par exemple d'éliminer la dépendance envers un nœud pour pouvoir effectuer une maintenance. De tels outils pourraient aussi fournir les informations sur la charge mémoire et processeur de chacun des nœuds. Une telle information serait alors utile pour la mise en place d'un ordonnanceur pouvant automatiquement distribuer les microprocessus sur un ensemble de nœuds disponibles, dans le but de répartir la charge.

Un point intéressant est que les canaux de communication hybrides éliminent la nécessité de créer soi-même un protocole de communication entre les différents systèmes. Le programmeur peut donc se concentrer sur la structure globale de son application distribuée et *dstackless* s'occupe du reste. À ce niveau, il serait intéressant d'avoir une implémentation très performante de la communication entre les nœuds.

L'expérimentation nécessaire pour connaître la performance de *dstackless* a volontairement été mise de côté dans le cadre de ce projet. Nous voulions tout d'abord définir un modèle simple d'utilisation et ensuite implémenter un prototype permettant de vérifier les fonctionnalités décrites dans ce mémoire. Dans la première version de *dstackless*, le protocole de communication entre les nœuds a été implémenté à partir de zéro, sans soucis de performance. Des tests de performance permettraient de comparer cette première implémentation aux autres outils offerts sur le marché. Par exemple, l'échange d'objets sur les canaux pourrait être comparé à MPI pour ce qui est de l'efficacité du transfert des objets ou l'accès aux objets distribués pourrait être comparé aux autres outils offrant des mémoires distribuées. Par rapport à ces tests, une amélioration des performances de *dstackless* impliquerait potentiellement l'intégration avec un ou plusieurs de ces outils.

La sécurité a aussi été écartée lors de la conception du module *dstackless*. Ce point devient très important pour que le module puisse être utilisé dans un environnement non protégé. Le niveau de sécurité nécessaire doit comprendre au minimum la validation de la source des microprocessus reçus, un chiffrement des canaux de communication et un système de droits d'accès permettant de limiter ce qu'un microprocessus peut faire sur un système donné. Tous ces points pourront être étudiés dans de futurs travaux de recherche.

Bibliographie

[1] What's all this fuss about Erlang ? http://pragmaticprogrammer.com/articles/erlang. [Visité le 8 janvier 2011].

[2] About Erlang. http://www.erlang.org/about.html. [Visité le 8 janvier 2011].

[3] J. Armstrong. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[4] C. Berge. *Théorie des graphes et ses applications*. Collection Univesitaire des Mathématiques, Dunod, Paris, 1958.

[5] L. Cardelli and A. D. Gordon. Mobile Ambients. In M. Nivat, editor, *FoSSaCS*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[6] J. Dean and S. Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. In *OSDI '04*, pages 137–150, 2004.

[7] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *23rd ACM Symposium on Principles of Programming Languages*, January 1996.

[8] G. Germain. Concurrency oriented programming in termite scheme. In *ERLANG '06 : Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 20–20. ACM, 2006.

[9] C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents : Are they a good idea ? *MOBILE OBJECT SYSTEMS TOWARDS THE PROGRAMMABLE INTERNET*, 1222(1997) :25–45, 1997.

[10] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[11] IronPython. http://ironpython.codeplex.com/. [Visité le 26 septembre 2010].

[12] The Jython Project. http://www.jython.org/. [Visité le 26 septembre 2010].

[13] B. Lewis and D. J. Berg. *Threads primer : a guide to multithreaded programming*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1995.

[14] R. Milner. *A Calculus of Communication Systems*. Springer, 1980.

[15] R. Milner. *The Polyadic pi-Calculus : A Tutorial*, 1993.

[16] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Part I. *Information and Computation*, 100(1) :1–40, Sept 1992.

[17] MPI Documents. http://www.mpi-forum.org/docs/. [Visité le 19 avril 2011].

[18] Interprocess Communications (Windows). http://msdn.microsoft.com/en-us/library/aa365574(VS.85).aspx, 2009. [Visité le 26 septembre 2010].

[19] L. Palmer. undo() ? http://groups.google.com/group/perl.perl6.language/msg/b0cfa757f0ce1cfd?pli=1, June 2004. [Visité le 26 septembre 2010].

[20] Parallel python - Home. http://www.parallelpython.com/, 2010. [Visité le 5 octobre 2010].

[21] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[22] PyPy. http://pypy.org/. [Visité le 26 septembre 2010].

[23] PyPy JIT. http://codespeak.net/pypy/dist/pypy/doc/jit/index.html. [Visité le 26 septembre 2010].

[24] PyPy objspace-proxies. http://codespeak.net/pypy/dist/pypy/doc/objspace-proxies.html. [Visité le 26 septembre 2010].

[25] PyPy stackless. http://codespeak.net/pypy/dist/pypy/doc/stackless.html. [Visité le 26 septembre 2010].

[26] The Python Language Reference - Python Documentation. http://docs.python.org/reference/index.html. [Visité le 21 décembre 2010].

[27] sys.stdin, sys.stdout, sys.stderr - Python Documentation. http://docs.python.org/library/sys.html#sys.stdin. [Visité le 26 septembre 2010].

[28] General Python FAQ - Python Documentation. http://docs.python.org/faq/general. [Visité le 18 avril 2011].

[29] Python Programming Language - Official Website. http://www.python.org/. [Visité le 26 septembre 2010].

[30] D. Reilly. Mobile agents - process migration and its implications. http://www.davidreilly.com/topics/software_agents/mobile_agents/, 1998. [Visité le 5 octobre 2010].

[31] D. Richie. The evolution of the unix time-sharing system. *AT&T Bell Laboratories Technical Journal*, 63(6 Part 2) :1577–93, October 1984.

[32] S.-W. Ryu and B. C. Neuman. Garbage collection for distributed persistent objects. In *Workshop on Compositional Software Architectures*, January 1998.

[33] D. Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '93, pages 151–166, London, UK, 1993. Springer-Verlag.

[34] R. Schollmeier. [16] a definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 101–, Washington, DC, USA, 2001. IEEE Computer Society.

[35] Stackless Python - About Stackless. http://www.stackless.com/. [Visité le 26 septembre 2010].

[36] Stackless Python - Pickling. http://www.stackless.com/wiki/Pickling. [Visité le 26 septembre 2010].

[37] W. Stallings. *Operating Systems : Internals and Design Principles*. Prentice Hall Press, Upper Saddle River, NJ, USA, 6th edition, 2008.

[38] The JoCaml System (old website). http://moscova.inria.fr/oldjocaml/index.shtml, 2005. [Visité le 26 septembre 2010].

[39] The JoCaml System. http://jocaml.inria.fr/, 2010. [Visité le 26 septembre 2010].

[40] ipc. http://opengroup.org/onlinepubs/007908775/xsh/ipc.html, 1997. [Visité le 26 septembre 2010].

[41] fork. http://www.opengroup.org/onlinepubs/000095399/functions/fork.html, 2004. [Visité le 26 septembre 2010].

[42] pipe. http://www.opengroup.org/onlinepubs/9699919799/functions/pipe.html, 2008. [Visité le 26 septembre 2010].

[43] C. Tismer. Continuations and Stackless Python. http://www.stackless.com/spcpaper.htm, 1999. [Visité le 26 septembre 2010].

[44] TOP-C Home Page. http://www.ccs.neu.edu/home/gene/topc.html. [Visité le 19 avril 2011].

[45] unladen-swallow. http://code.google.com/p/unladen-swallow/. [Visité le 26 septembre 2010].

[46] Berkeley Unified Parallel C (UPC) Project. http://upc.lbl.gov/. [Visité le 19 avril 2011].

[47] J. van Rossum and P. Moore. Pep 302 – New Import Hook. http://www.python.org/dev/peps/pep-0302/, 2002. [Visité le 26 septembre 2010].

[48] P. Welch, J. Foster, and J. Aldous. Communicating Sequential Processes for Java (JCSP). http://www.cs.kent.ac.uk/projects/ofa/jcsp/, 2002. [Visité le 26 septembre 2010].

[49] J. White. Mobile agents white paper, 1996.

[50] Wikipedia - Fiber. http://en.wikipedia.org/wiki/Fiber_(computer_science). [Visité le 26 septembre 2010].

[51] D. Zöbel. The deadlock problem : a classifying bibliography. *SIGOPS Oper. Syst. Rev.*, 17 :6–15, October 1983.