Université Joseph Fourier — Master 2 Recherche - Systèmes et Logiciels

---

Gestion autonome de flots d'exécution événementiels

Projet réalisé par :
Fabien GAUD

Soutenu le :
18 juin 2007

Équipe Sardes
Inria Rhône-Alpes - Laboratoire d'Informatique de Grenoble

Encadrants :
Renaud Lachaize
Vivien Quéma

JURY :

Laurent Besacier	, Membre du jury permanent
Rachid Echahed	, Membre du jury permanent
Jean-Marc Vincent	, Membre du jury permanent
Yves Denneulin	, Examinateur externe
Renaud Lachaize	, Encadrant
Vivien Quéma	, Encadrant

Remerciements

Je tiens à remercier tout d'abord l'équipe Sardes de m'avoir accueilli et d'avoir apporté le financement nécessaire à ces travaux. Je tient également à apporter toute ma gratitude Renaud et Vivien pour leur suivi, leur disponibilité et leur conseils avisés. Je remercie également Natacha, tous mes amis et ma famille qui ont participé à la relecture de ce rapport.

Table des matières

Table des figures

Introduction

Problématique

Ce projet s'est déroulé dans l'équipe Sardes qui fait partie de l'INRIA[1] et du LIG[2]. Le projet Sardes s'intéresse à la définition de technologies logicielles pour la construction de systèmes adaptables. L'un des axes de travail de cette équipe porte sur la définition de méthodes de développement d'applications performantes. Ce sujet est l'objet de nombreuses recherches depuis plusieurs années. Deux modèles de programmation prédominent actuellement :

- **La programmation à base de processus légers (threads)** qui consiste à utiliser plusieurs flots d'exécution concurrents au sein d'une même application.
- **La programmation événementielle** qui consiste à utiliser un seul flot d'exécution traitant des événements.

Pour différentes raisons que nous détaillerons dans la suite de ce rapport, ces deux types de programmation ne sont pas satisfaisants. D'autres approches ont été proposées dont notamment un modèle hybride basé sur la notion d'étages. L'application est découpée en différents étages qui communiquent de manière asynchrone à l'aide de files de messages. Chaque étage dispose de ses propres ressources d'exécution. Ce modèle permet d'obtenir de bonnes performances, notamment dans le cas d'applications de serveurs de données. Par ailleurs, cette notion d'étage est très intéressante puisqu'elle introduit aussi des notions de modularité ainsi que de reconfigurabilité.

Néanmoins, la programmation à base d'étages a des limitations liées au fait que l'interaction entre les étages est effectuée systématiquement de manière asynchrone. Or, si dans certains cas il est plus intéressant que l'exécution soit effectuée en asynchrone — ce type d'exécution apportant des garanties de résistance à la charge —, il est néanmoins parfois préférable que les étages intéragissent de façon synchrone — les interactions synchrones permettent notamment d'améliorer les performances à basse charge. Il serait donc intéressant de disposer d'un modèle de programmation par étage permettant de changer dynamiquement de modèle d'exécution (synchrone ou asynchrone).

Contributions

Nous présentons, dans ce rapport, les travaux qui ont été effectués dans le but de proposer une évolution de l'architecture à étages. L'objectif de ces travaux est de permettre au support d'exécution de choisir entre un modèle d'exécution asynchrone et un modèle d'exécution synchrone. Ce choix doit pouvoir être dynamique dans le but de s'adapter aux paramètres

[1] Institut National de Recherche en Informatique et en Automatique
[2] Laboratoire d'Informatique de Grenoble

d'exécution (charge, type de support, etc.). Les travaux qui sont présentés dans ce rapport sont les suivants :

1. Proposition d'une architecture adaptable dynamiquement aux deux modèles d'exécution. Cette architecture est basée sur SEDA, un projet qui à notamment introduit le concept de programmation par étages.
2. Vérification qu'il est intéressant d'avoir soit une exécution synchrone soit une exécution asynchrone selon les cas. Cette vérification a été effectuée au moyen de micro-tests sur des architectures simples. Nous nous sommes ici principalement intéressés au monde des serveurs de données, tels que les serveurs web.
3. Proposition de différents paramètres qui peuvent influencer le choix du modèle d'exécution.
4. Mise en œuvre d'un mécanisme d'adaptation dynamique en fonction de l'évolution des paramètres précédemment cités.

Organisation du document

Ce rapport est organisé comme suit. La première partie est consacrée à l'étude des solutions existantes qui permettent de programmer des applications performantes. Nous passons notamment en revue les avantages et les inconvénients des solutions à base de threads et d'événements ainsi que les améliorations apportées à ces solutions pour corriger leur défauts. Enfin, nous présentons une nouvelle façon de programmer des applications à étages.

Dans une seconde partie nous mettons en évidence les manques actuels de la solution à base d'étages et expliquons les objectifs de notre contribution. Dans une troisième partie, nous présentons l'évolution de l'architecture à étages proposée et décrivons la réalisation de son implantation. Enfin, nous donnons enfin les résultats des expériences menées sur différents types d'applications.

Chapitre 1

État de l'art

Ce chapitre dresse un inventaire des différentes techniques existantes pour la programmation d'applications performantes. Deux techniques opposées se détachent notamment. La première consiste à utiliser plusieurs processus par application. Ces processus peuvent être soit des processus lourds, soit des processus légers (threads). La seconde méthode consiste à programmer l'application en utilisant des événements.

Dans cette partie, nous allons faire une présentation des solutions existantes pour programmer des applications parallèles. Tout d'abord, nous allons notamment voir les avantages et les inconvénients de la programmation par processus (essentiellement les processus légers même si les processus lourds seront abordés). Dans une seconde partie, nous discuterons des avantages et des inconvénients de la programmation événementielle. Enfin nous verrons comment des solutions ont été proposées pour arriver à concilier les avantages de ces deux techniques tout en essayant d'en éliminer les défauts.

1.1 Programmation concurrente par processus

La programmation concurrente par processus est le modèle de programmation le plus utilisé actuellement pour concevoir des applications parallèles performantes. Dans cette section, nous allons dans un premier temps décrire les principes de la programmation par processus. Nous en verrons ensuite ses points forts et ses points faibles.

1.1.1 Processus lourds / Processus légers

Principes

Un processus peut être défini comme une instance d'un programme. Lorsqu'un processus se lance, le système d'exploitation lui accorde une quantité d'espace mémoire. Cet espace va contenir notamment :

- Les informations nécessaires au programme (instructions, variables, ...)
- La pile du processus. Cette pile est notamment utilisée pour sauvegarder les variables lors d'appels de fonction ou de commutation de processus
- D'autres informations telles que l'état du processus.

Il est important de voir que chaque instance du programme va donner naissance à un processus qui possédera toutes ces informations. Les processus légers, comme nous le verrons permettent de partager certaines de ces informations. Dans un système multi-tâches, plusieurs

instances de processus sont lancées simultanément. Pour savoir quels sont les processus à exécuter, le système affecte un état à chaque processus. La figure 1.1 illustre les différents états dans lesquels un processus peut passer. Il est a noter que ces états peuvent varier d'un système d'exploitation à un autre. Sur un système mono-processeur, un seul processus peut

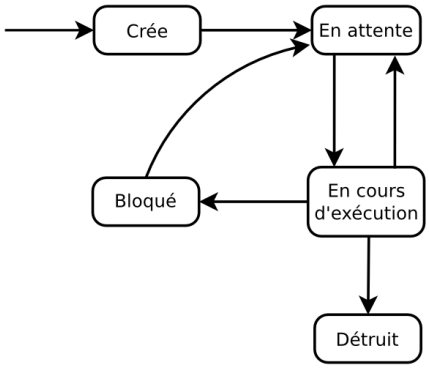

FIG. 1.1 – Différents états d'un processus

être actif à un instant donné. De même, sur un système possédant k processeurs, il y a k processus actifs à un moment donné. Pour donner une impression de concurrence, le système ordonnance les processus en leur affectant un quantum de temps. L'ordonnancement peut aussi être influencé en affectant une priorité aux processus.

Les changements de contexte interviennent lors de la fin d'un quantum de temps. Le système d'exploitation va sauvegarder les différentes informations constituant le contexte du processus (notamment l'état des registres), mettre le processus courant dans l'état "en attente", charger les informations d'un autre processus et enfin mettre le nouveau processus dans l'état "en cours". L'opération de changement de contexte est donc très coûteuse puisqu'elle nécessite des accès à la mémoire (généralement plus coûteux que des opérations de calcul) et le passage par le mode noyau. Le réglage du quantum de temps est donc très important. En effet, si ce dernier est trop court, le coût des changements de contexte devient trop important. A l'inverse, s'il est trop long, le système n'est pas assez réactif.

Comme nous l'avons vu précédemment, chaque processus dispose de son propre espace mémoire. La technique pour créer des processus à l'intérieur d'un processus, sur la plupart des systèmes, est l'utilisation de l'appel système *fork*. Dans ce cas, le système alloue un nouvel espace mémoire et effectue une recopie du processus courant. Après un appel à fork, les deux processus obtenus sont identiques. Cet appel système est donc très coûteux.

Il n'est pas toujours souhaitable que deux processus soient dans des espaces d'adressage différents. En effet, il peut être nécessaire de partager certaines données entre les processus. Avec les processus lourds, la communication se fait grâce à des canaux particuliers (IPC[1]) dont le coût est important.

Les processus légers (ou *threads*) ont été créés pour réduire ces coûts tout en conservant le parallélisme. Chaque thread dispose de sa propre pile mais partage la plupart de ses informations avec les autres threads du processus. Un autre intérêt est que le coût (en temps) des changements de contexte entre deux processus légers est moindre accéléré puisque seules

[1]Inter Processus Communication

certaines parties du contexte (pile, registres, ...) devront être rechargées. Toutefois ce partage de données peut poser des problèmes s'il n'est pas bien protégé, comme nous le verrons plus loin. Dans la suite de ce document, nous utiliserons l'expression *threads* pour désigner les processus légers et *processus* pour désigner les processus lourds.

Il existe deux types de threads : ceux de niveau noyau et ceux de niveau utilisateur. Les threads de niveau utilisateur, illustrés par la partie a) de la figure 1.2, ne sont pas visibles par le système d'exploitation. En effet, ces threads vont être gérés et ordonnancés par une bibliothèque au niveau utilisateur, qui va notamment effectuer les diverses opérations de leur cycle de vie (création, destruction, activation, ...). La solution est donc facilement portable sur différents systèmes d'exploitation. Cependant, les appels systèmes bloquants doivent être réécrits. En effet, ces appels doivent obligatoirement passer par la bibliothèque de threads. Dans le cas contraire, l'ordonnanceur n'aurait pas connaissance du blocage d'un processus et ne pourrait donc pas suspendre son ordonnancement. Enfin, comme le système n'a pas connaissance des threads de niveau utilisateur, il n'est pas possible de tirer parti du parallélisme de l'architecture matérielle. Dans le cas des threads de niveau noyau, partie b) la figure 1.2, l'ordonnanceur est géré par le noyau ce qui permet d'exploiter les architectures multiprocesseurs. Cependant, la gestion de ces threads est plus coûteuse que celle des threads de niveau utilisateur car elle nécessite des passages en mode noyau. Pour remédier à ces différents inconvénients, une solution hybride peut être utilisée. Elle consiste à associer N threads utilisateurs répartis sur P threads noyau (avec $N \geq P$). Cette association est illustrée par la partie c) de la figure 1.2.

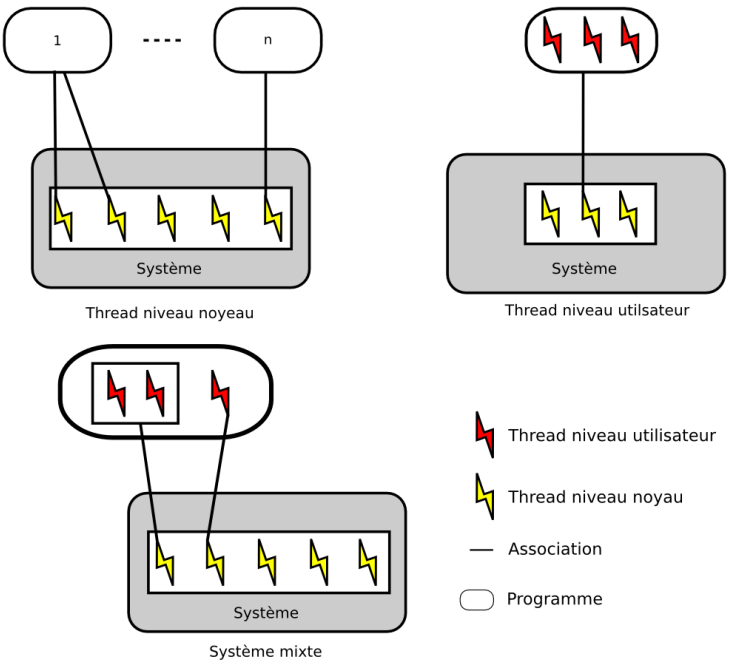

FIG. 1.2 – Threads noyau / threads utilisateurs

Exemple typique

La programmation par threads est probablement la plus utilisée actuellement pour obtenir de la concurrence. Prenons par exemple le cas d'un serveur, dont le comportement correspond au pseudo-code ci-dessous.

```
int main(){
  while(true){
    socket s = attendre(connexion);
    initialiser_tread(s);
    démarrer_thread();
  }
}
```

Ce programme serveur est basé sur une boucle pour récupérer les connexions et sur des threads pour faire les traitements associés. Le thread principal est chargé d'accepter les connexions et délègue ensuite la communication la communication avec le client à un autre thread. Ainsi, comme un thread est lancé pour chaque connexion, il est possible de traiter plusieurs connexions simultanément.

1.1.2 Points faibles

Dans cette partie, nous décrivons les défauts de la programmation à base de threads. De nombreux problèmes découlent notamment de l'utilisation de mémoire partagée entre les threads. Nous allons énumérer les principaux problèmes induits par cette manière de programmer, qui peuvent soit conduire à des problèmes de performance, soit à des problèmes de correction.

Synchronisation

Comme nous l'avons vu dans la partie 1.1.1, une des caractéristiques des threads est de pouvoir partager de la mémoire. Cette caractéristique est celle qui peut potentiellement engendrer le plus de problèmes.

Le premier est celui de l'incohérence des données. En effet, si les données ne sont pas protégées, il est possible que le résultat de l'exécution de deux threads manipulant la même donnée puisse être erroné. Comme l'explique Lee [15], ce problème vient du fait que l'exécution des threads est non déterministe. En effet, c'est l'ordonnanceur du système qui va choisir le quantum de temps alloué à chaque thread. Ce quantum de temps peut ne pas être identique d'une exécution à l'autre. De plus, comme la plupart des systèmes sont préemptibles, le système d'exploitation peut à tout moment décider d'interrompre l'exécution d'un processus pour traiter une E/S[2]. Ainsi, les problèmes qui apparaissent sont liés à la préemption et non pas à l'exécution des threads eux-mêmes [6]. Un exemple typique d'incohérence est la gestion d'un compte bancaire. Un thread va débiter le compte et donc exécuter :

```
1. solde = getCompte(numCompte);
2. solde = solde - valeur;
3. updateCompte(numCompte, solde);
```

[2]Entrée/Sortie

Un autre thread va simultanément ajouter de l'argent sur le compte :

```
4. solde = getCompte(numCompte);
5. solde = solde + valeur;
6. updateCompte(numCompte, solde);
```

Supposons que le solde soit à 10 initialement, que le thread 1 ajoute 2 et que le thread 2 enlève 3. Si l'exécution est `1,2,3,4,5,6`, le résultat obtenu est 9, ce qui est le résultat attendu. Par contre, si l'exécution effectuée est `1,4,2,3,5,6`, le résultat obtenu est 7. On voit donc dans ce cas que le résultat n'est pas cohérent et que l'exécution n'est pas déterministe, puisque deux exécutions donnent potentiellement deux résultats différents.

La solution la plus généralement choisie est de verrouiller les parties du code manipulant des variables partagées, c'est à dire de garantir qu'un seul thread manipulant ces données ne peut être exécuté à un moment donné. Cela est possible grâce à l'utilisation de verrous. Toutefois cette solution amène d'autres problèmes. Tout d'abord, le coût du verrouillage de parties de code est important. En effet, il faut passer par le mode noyau pour éventuellement endormir un processus en attente de la section critique, puis réveiller les processus en attente à la sortie. Cela implique aussi de gérer une liste des processus en attente. D'autre part, un autre problème important est celui des interblocages (*deadlocks*). Dans une situation d'interblocage, chaque processus attend un verrou qu'un autre processus possède. Cela peut se produire lorsque des processus prennent des verrous dans un ordre différent. Un problème similaire est celui de boucle sans progrès (*livelock*). Dans ce cas, le système n'est pas bloqué car les threads essaient de prendre des verrous (et en relâchent périodiquement) mais les threads ne progressent pas car ils n'arrivent pas à acquérir tous les verrous dont ils ont besoin.

Les problèmes d'interblocages soulèvent une autre difficulté [22] : comment arriver à composer un programme avec des bibliothèques développées par des tierces parties ? En effet, on ne peut pas être sûr, à moins d'analyser le code de la bibliothèque, que cette dernière ne va pas prendre un verrou sur une donnée verrouillée par le programme appelant et ainsi être dans une situation d'interblocage. Cela limite donc la réutilisabilité du code qui est un critère important actuellement.

Coûts

Un autre reproche fait à la programmation par threads concerne son passage à l'échelle. En effet, plusieurs types de coûts sont induits tels que des coûts de synchronisation, des coûts de création/destruction de thread, des coûts de changement de contexte, des coûts mémoire dûs à la pile des threads, qui est généralement de taille fixe, ou encore des coût dûs à la mauvaise utilisation des caches.

Nous avons déjà parlé des coûts de synchronisation dans la partie précédente. La création et la destruction d'un thread ont un coût non négligeable au niveau du système. Cependant, ces coûts peuvent être amortis si l'opération effectuée est conséquente. La granularité des tâches associées aux threads a donc un impact sensible.

Le changement de contexte aussi à un coût très important. En effet, lors d'un changement de contexte, le système d'exploitation doit effectuer la sauvegarde de tout ce qui va être utile pour pouvoir redémarrer le processus dans le même état (registres, descripteurs de fichiers, ...) ainsi que la restauration du contexte propre au nouveau processus en exécution. Ces opérations nécessitent de nombreux accès à la mémoire ainsi qu'un passage en mode noyau.

La taille de la pile est un reproche souvent fait à la programmation par threads. En effet, dans les systèmes d'exploitation, lors de la création d'un thread, le système alloue en général une pile de 8Ko pour les threads noyau et de plusieurs méga-octets pour les threads utilisateur. Cette place n'est pas toujours utilisée par le processus, excepté dans le cas de fonctions récursives. Cela limite aussi le nombre de threads pouvant être alloués par le système et, dans le cas de serveurs, peut s'avérer gênant.

Enfin, la programmation par threads entraîne une mauvaise utilisation des caches du processeur[3]. En effet, le processeur va exécuter une partie du code d'un thread, l'interrompre et exécuter le code d'un autre thread qui n'utilise pas forcement les mêmes données. Le système va donc charger les informations dans le cache pour le nouveau thread en écrasant éventuellement les parties de mémoire chargées pour le premier, comme les caches sont de faible taille. Un fréquence élevée de changements de contexte va donc faire baisser le taux d'utilisation des caches.

Autres considérations

Dans le cas de la programmation par threads, deux points supplémentaires peuvent être gênants : la difficulté pour déboguer ainsi qu'une portabilité parfois limitée.

Déboguer des programmes multi-threadés peut s'avérer très difficile. Cela vient notamment du fait que l'exécution de l'application est dépendante de l'ordonnanceur du système. Des outils de vérification automatique existent. Ces outils se basant généralement sur les traces d'exécution, leur passage à l'échelle est difficilement réalisable [15].

La portabilité des applications est aussi parfois un problème. En effet, les threads noyau ne sont pas portables car tous les systèmes ne les proposent pas. Les threads utilisateurs, quant à eux, dépendent de la présence d'une bibliothèque sur le système. Une norme (threads *POSIX*) a été définie pour simplifier la portabilité, mais les différents systèmes la respectent plus ou moins. Seuls des langages de plus haut niveau tels que Java garantissent la portabilité puisqu'ils utilisent une machine virtuelle.

1.1.3 Points forts

Un des apports de la programmation par threads est qu'elle permet de tirer parti des architectures parallèles. Dans le cas de multiprocesseurs, plusieurs solutions peuvent être choisies, comme affecter un thread à chaque processeur ou démarrer de nombreux threads qui seront ordonnancés sur les différents processeurs. Cela permet au système d'être plus réactif puisque plus d'opérations pourront être traitées à la fois.

De plus, ce modèle conserve l'aspect séquentiel des programmes. Le découpage en différents processus s'effectue de manière plus naturelle en les parties qui peuvent être exécutées simultanément.

[3]Ces caches matériels sont de petites zones de mémoire à accès rapide placées près du processeur. Il y a plusieurs niveaux de cache qui diffèrent par leur taille et leur temps d'accès. Par exemple le cache de niveau 1 est situé le plus près du processeur et a un temps d'accès très court mais une capacité très faible. Le cache de niveau 2 est un peu plus grand mais est aussi un peu moins rapide. Les caches ont une utilité du fait que les temps d'accès à la mémoire centrale sont très longs, entraînant la perte de nombreux cycles de processeur.

1.2 Programmation concurrente par événements

Les risques et le coûts précédemment cités ont conduit Ousterhout [18] à recommander d'éviter l'usage de threads au profit de la programmation événementielle. La programmation événementielle est un autre type de programmation permettant de construire des applications performantes. Nous allons voir dans la suite les principes de l'utilisation d'événements ainsi que les points forts et points faibles de cette méthode.

1.2.1 Principes

La programmation événementielle est organisée autour d'une boucle de contrôle et de traitants d'événements. Le programme enregistre des traitants pour certains événements. Lorsque la boucle de contrôle détecte l'arrivée d'un événement, elle exécute le traitant associé. La figure 1.3 illustre les principes de la programmation événementielle. Cette figure représente l'arrivée d'un événement et l'appel au traitant par la boucle de contrôle.

Le domaine d'application des événements le plus connu est le domaine des interfaces graphiques. En effet, l'interface va réagir à des événements tels que le mouvement d'un dispositif de pointage. Cependant la programmation événementielle est aussi utilisée pour la mise en œvre de serveurs à haute disponibilité tels que le répartiteur de charge HAProxy [1] ou le serveur web Zeus [4].

Pour que ce type de programmation soit efficace il faut à la fois que les traitants soient courts et que les opérations faites dans les traitants soient non bloquantes. En effet, si les opérations sont longues, comme il n'y a qu'un seul processus, le système n'est plus réactif. Ceci est dû au fait qu'un seul évènement est traité à la fois. De façon analogue, si une opération bloquante est effectuée dans un traitant (comme par exemple une E/S synchrone), la réactivité de l'application va être fortement réduite. Pour pouvoir utiliser les événements de façon optimale, il faut donc disposer d'E/S asynchrones. Ainsi, le programme ne reste pas bloqué en attente d'une E/S et revient dans la boucle de contrôle. À la fin de l'E/S, la boucle de contrôle est avertie et appelle le traitant associé.

1.2.2 Points faibles

Nous présentons dans cette section les reproches qui sont généralement faits à la programmation événementielle.

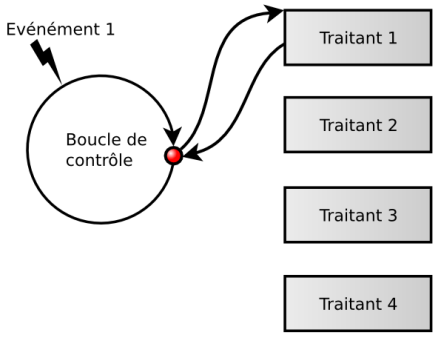

Fig. 1.3 – Principes de la programmation événementielle

La difficulté de programmation est le défaut principal de la programmation événementielle. En effet, contrairement à la programmation à l'aide de threads qui conserve l'aspect séquentiel de la programmation traditionnelle, une application développée de manière événementielle est structurée en une multitude de traitants. Il est donc plus difficile pour le programmeur d'avoir une vision globale de l'application puisqu'une exécution n'est pas vue de manière séquentielle mais comme un ensemble d'exécutions de traitants. Cette situation peut éventuellement engendrer plus d'erreurs de programmation, du fait de l'inexpérience dans ce domaine.

L'utilisation de multiples traitants pose un autre problème. En effet, il est difficile pour les traitants de se faire passer des informations du fait qu'il y a toujours un retour à la boucle de contrôle. Similairement, le traitement des exceptions peut poser des problèmes puisqu'il n'est pas possible de faire remonter l'exception à la procédure qui a enregistré le traitant. Une autre difficulté importante liée à l'utilisation de nombreux traitants est la gestion de la mémoire. En effet, si une zone de mémoire est allouée par un traitant, il peut parfois être difficile de savoir quand la libérer. Cette situation peut entraîner des erreurs d'accès à la mémoire ou des problèmes de fuites de mémoire.

Une autre difficulté est la nécessité d'avoir des E/S asynchrones. Comme expliqué précédemment, cela est nécessaire pour garantir la réactivité de l'application. Si les E/S réseau asynchrones sont en général bien supportées, les E/S asynchrones sur des périphériques tels que les disques durs ne sont que rarement disponibles. Comme nous le verrons dans la partie 1.3.1, certains systèmes implémentent donc une solution à base de threads pour permettre d'utiliser des E/S asynchrones sur les systèmes d'exploitation ne le proposant pas.

Enfin, un autre point faible important de cette méthode est qu'elle ne permet pas d'utiliser efficacement les multiprocesseurs. Ceci est dû au fait que le déroulement du programme n'est effectué que dans un seul et unique flot d'exécution. Ainsi il ne pourra y avoir qu'un seul traitant en activité à un moment donné, ce qui ne permet pas d'utiliser les capacités des multiprocesseurs pour le programme.

1.2.3 Points forts

L'un des point fort de la programmation événementielle est que le développeur est déchargé de la gestion de la concurrence sur les variables partagées puisqu'un seul thread peut être actif à un instant donné. Cette caractéristique permet aussi à l'application d'être plus performante. Ceci est principalement dû aux économies de verrouillage des données, de changements de contexte et de création/destruction de threads. Le fait de n'avoir qu'un seul processus permet aussi une bonne montée en charge et le traitement de nombreuses requêtes simultanées. En effet, l'occupation mémoire est celle d'un seul processus. Enfin les solutions obtenues sont plus portables que celles utilisant des threads puisqu'elles ne dépendent pas des capacités du système d'exploitation. La seule exception concerne l'utilisation d'E/S asynchrones, mais ces dernières peuvent être émulées (comme expliqué en section 1.3.1).

1.3 Programmation concurrente hybride

L'objectif de la programmation concurrente hybride est de mélanger la programmation par threads et la programmation événementielle pour obtenir une solution combinant leurs avantages respectifs. Dans cette partie, nous allons voir les différents projets en rapport avec

la programmation hybride, chacun apportant une solution à un problème précis évoqué précédemment.

1.3.1 Flash

L'objectif de Flash [19] est de pouvoir proposer une alternative aux architectures classiques pour les serveurs web. La programmation événementielle est une bonne solution pour les serveurs web puisqu'elle permet une bonne montée en charge. Cependant, le problème de la disponibilité d'E/S asynchrones dans le système d'exploitation doit être résolu. En effet, une E/S bloquante entraîne un surcoût inacceptable pour un serveur web. Flash s'attaque donc au problème des E/S asynchrones. Pour cela, une nouvelle architecture à été proposée : l'architecture AMPED[4]. La programmation est toujours de type événementielle. A chaque itération, la boucle de contrôle vérifie s'il y a un événement à traiter. Dans un tel cas, le traitant correspondant est appelé. C'est à ce moment là que la distinction intervient, selon que le traitant effectue une E/S disque bloquante ou une opération non bloquante. Dans le deuxième cas, le traitant finit ses opérations et retourne à la boucle de contrôle. Comme l'illustre la figure 1.4, dans le premier cas, le traitant démarre un processus *Processus1* qui gère cette E/S et enregistre un nouveau traitant pour la fin de cette E/S. Ainsi, seul le processus gérant la communication avec le disque est bloqué en attente de la fin de l'opération.

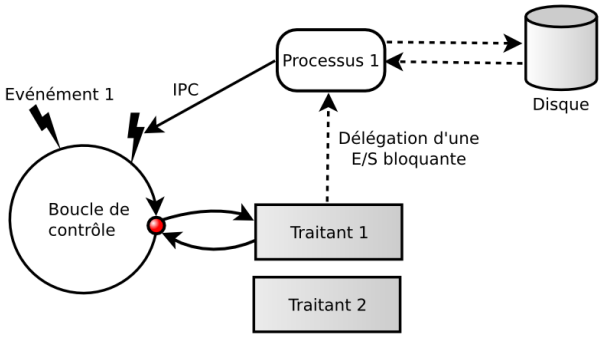

FIG. 1.4 – L'architecture AMPED

Le premier problème soulevé est de savoir comment informer la boucle de contrôle de la fin d'une opération d'E/S. Pour cela, Flash utilise les IPC (voir section 1.1.1). Lorsque l'E/S est finie et que le processus termine, il va communiquer, au moyen des IPC, avec la boucle de contrôle pour générer un événement. Le deuxième problème est de savoir si la donnée n'était pas en cache. En effet, dans ce cas là, il n'est pas utile de traiter l'E/S dans un processus séparé. Pour cela, Flash utilise l'opération `mincore` qui permet de tester si une donnée est en mémoire centrale ou non. Dans le premier cas, le traitement de l'E/S est poursuivi dans le traitant. Sinon, un nouveau processus est utilisé pour émuler une E/S asynchrone.

1.3.2 Libasync-smp

Le projet Libasync-smp [27] vise à étendre la programmation événementielle aux architectures multiprocesseurs et se base sur Libasync [10], une bibliothèque développée pour sim-

[4]Asymetric Multi-Process Event-Driven

plifier la programmation événementielle. Nous allons expliquer, brièvement, dans un premier temps les buts recherchés par Libasync. Dans une seconde partie, nous verrons comment cette bibliothèque a évolué pour supporter une exécution sur plusieurs processeurs.

Libasync

Cette bibliothèque, écrite en C++, a été conçue pour répondre aux différents problèmes soulevés par l'utilisation de la programmation événementielle. Comme expliqué dans la partie 1.2.2, la difficulté de la programmation événementielle est souvent l'une des raisons principales de sa faible adoption. En effet, les tâches nécessaires sont répétitives et le programme est une succession d'appels à des traitants dont les interactions ne sont pas faciles à déceler. Par ailleurs, les mécanismes de sauvegarde et de récupération de contexte sont aussi à la charge du programmeur.

Pour répondre à ce problème, Libasync propose plusieurs améliorations. La première est une boucle de contrôle générique qui récupère un événement et effectue l'appel au traitant associé. Le fait que la boucle de contrôle soit inclue dans la bibliothèque décharge le programmeur de sa gestion qui est souvent répétitive et fastidieuse. La boucle de contrôle de Libasync utilise l'appel `select` pour récupérer un nouvel événement. Elle dispose de méthodes pour enregistrer/supprimer un couple (événement, traitant). L'utilisation de modules prédéfinis pour effectuer certaines séquences d'événements répétitives permet aussi de simplifier la tâche du programmeur. Par exemple, dans le cas d'un appel DNS, plusieurs événements vont être gérés automatiquement ce qui permet d'alléger le code du programme.

Un autre problème de la programmation événementielle est de savoir comment partager des informations entre les traitants. Pour cela, Libasync dispose d'une méthode appelée *wrap*, qui permet d'enregistrer des arguments à passer au traitant lors de son appel par la boucle de contrôle. Par exemple, si on a `f2 = wrap(f1,var1)` et qu'on enregistre f2, l'appel effectué par la boucle de contrôle sera `f1(var1)`, si elle ne passe pas d'arguments supplémentaires, ou `f1(var1,var2,...)` si elle ajoute des arguments.

Enfin, concernant le problème de la gestion de la mémoire dans un programme événementiel, Libasync inclut un ramasse-miettes (ou *garbage collector*) permettant de libérer le programmeur de cette gestion. Le ramasse-miettes regarde périodiquement quels sont les objets qui peuvent être enlevés de la mémoire, par exemple en comptant le nombre de références pointant sur l'objet. Le programmeur ne s'occupe donc plus de la libération de la mémoire, mais uniquement de son allocation.

Libasync-smp

Libasync-smp [27] étend la bibliothèque Libasync en permettant le support d'architectures parallèles. Le principe de la solution repose sur l'attribution de "couleurs" aux traitants lors de l'enregistrement auprès de la boucle de contrôle. Cette affectation est à la charge du programmeur. Les couleurs sont représentées par un entier sur 32 bits, ce qui laisse une grande plage de combinaisons possibles. Un traitant est décoré par une et une seule couleur. Le système garantit que deux traitants ayant la même couleur ne s'exécuteront pas en parallèle. En revanche, deux traitants n'ayant pas la même couleur peuvent s'exécuter en parallèle. Les traitants manipulant les mêmes données sont typiquement de la même couleur. La compatibilité avec la bibliothèque précédente est également assurée. En effet, si le traitant qui s'enregistre n'a pas choisi de couleur, une couleur par défaut lui est automatiquement affectée. Ainsi, par

défaut, toutes les traitants s'exécutent de façon séquentielle (car ils sont décorés de la même couleur). Cette méthode permet également à un programmeur de rajouter du parallélisme de manière incrémentale dans son application. Le parallélisme proposé par Libasync-smp est un parallélisme à gros grain puisqu'il n'est pas possible d'affecter plusieurs couleurs à un traitant. La synchronisation est donc de type exclusion mutuelle et il n'est pas possible d'implémenter des stratégies plus évoluées de type lecteur/rédacteur. Toutefois, les auteurs affirment que ceci est suffisant dans la majorité des cas. Une possibilité pour attribuer les couleurs est, par exemple, d'utiliser l'adresse de la variable partagée. Ainsi, les traitants manipulant cette variable auront la même couleur.

Une fonction importante du système est l'ordonnancement. Un programme est divisé en N threads, où N est le nombre de processeurs vus par le système d'exploitation. Libasync-smp utilise une règle simple : le traitant de couleur c est placé dans la file du thread $c \ mod \ N$. Cela permet de garantir le fait que les traitants de même couleur seront exécutés dans l'ordre de leur arrivée et de manière séquentielle. C'est également une bonne approche pour la gestion des caches des processeurs et pour obtenir une distribution de charge homogène. Pour chaque file, l'ordonnanceur doit choisir quel traitant va être exécutée. Pour cela, il privilégie les traitants de la couleur de celle qui vient de s'exécuter. Ces dernières manipulant souvent des variables partagées, cette optimisation permet de mieux utiliser les caches.

Pour améliorer l'utilisation de tous les processeurs, un thread qui a une file vide peut voler des traitants dans la file d'un autre thread. Dans ce cas, il doit également récupérer toutes les traitants de la même couleur, pour garantir que deux traitants de la même couleur ne sont pas exécutés simultanément. De ce fait, si une couleur est en cours d'exécution, les traitants de la même couleur ne peuvent pas être volés. Il faut aussi que les nouveaux traitants de la même couleur soit redirigées vers cette file.

L'évaluation a été effectuée en réalisant un serveur de fichiers cryptés. Celle-ci montre que Libasync-smp permet de tirer parti de la présence de plusieurs processeurs, notamment en raison du coût engendré par les opérations de cryptage.

1.3.3 StagedServer

Le projet StagedServer [13] vise à améliorer la programmation à base de threads. Un point important notamment reproché à cette dernière est la mauvaise utilisation des caches qui en résulte, comme expliqué dans la partie 1.1.2. Pour résoudre ce problème, deux principes vont être utilisés : l'ordonnancement par lot et la programmation par étages.

L'ordonnancement par lots

La mauvaise utilisation des caches est le résultat d'une séquence d'ordonnancement de threads ne manipulant pas les mêmes données. Plutôt que d'ordonnancer les threads de façon arbitraire, il vaut mieux regrouper et exécuter successivement les threads manipulant les mêmes données. En effet, si l'exécution du premier thread risque d'être coûteuse, puisqu'il faudra probablement rapatrier les données (les données de l'application et le code) depuis la mémoire centrale vers les caches, les exécutions suivantes bénéficieront du fait que les données sont déjà en cache.

Pour pouvoir créer des groupes de threads, il faut cependant pouvoir décrire quelles sont les opérations similaires. Une première solution est de comparer l'adresse de départ des threads, pour vérifier s'ils exécutent la même fonction. Cependant une telle observation se heurte au

fait que deux threads différents peuvent manipuler les mêmes données et perd ainsi de l'intérêt.

La programmation par étages

Pour permettre un regroupement des opérations similaires en groupes de threads, StagedServer utilise le principe de la programmation par étage, similaire à celui présenté dans la partie 1.3.4. L'exécution va être découpée en une succession d'étages communiquant entre eux. L'appel à un étage est fait de manière asynchrone, ce qui permet à l'appelant d'effectuer d'autres opérations après son appel. L'ordonnancement dans un étage est effectué par l'étage lui même. Ainsi, le support d'exécution va pouvoir par exemple décider si les opérations de l'étape doivent se faire en exclusion mutuelle ou s'il peut y avoir plusieurs exécutions simultanées. Une opération qui a commencé son exécution ne peut pas être arrêtée. L'intérêt d'utiliser la programmation par étages est que le découpage de l'application se fait naturellement puisque les comportements identiques vont être regroupés au sein d'un même étage. Ainsi, le problème de la reconnaissance des opérations similaires va être résolu puisqu'il s'agira en fait de traiter consécutivement une partie significative des événements d'un étage.

Pour améliorer encore les effets de cache, la solution qui a été choisie est de disposer de deux structures (une pile et une file) par étage et par processeur. Les événements générés par le processeur sur lequel est exécuté l'étage sont mis dans la pile. Les autres sont mis dans la file. Lorsque l'étage est ordonnancé, il commence par exécuter les opérations de la pile, bénéficiant ainsi encore plus des effets de cache. Puis il fini par exécuter les opérations de la file.

Les étapes sont organisées dans une liste par le programmeur. L'ordonnancement des étages sur les différents processeurs du système est effectué en parcourant cette liste d'abord en descendant, puis en remontant dans la liste des étages. Ce parcourt permet de bénéficier encore plus des effets de cache. Le processeur va ensuite exécuter les opérations de sa pile et de sa file. Il va ensuite choisir une nouvelle étape à traiter. Plusieurs schémas d'ordonnancement sont disponibles, et il est notamment possible de spécifier le fait qu'une étape peut être traitée par plusieurs processeurs (*shared stage*), ou par un seul (*exclusive stage*). Il est également possible de forcer l'ordonnancement d'un étage après un certain nombre d'opérations. Enfin il existe une communication entre les étages et ces derniers peuvent volontairement suspendre leur exécution pour favoriser un autre étage.

1.3.4 SEDA

Principes

SEDA [26] est une architecture conçue pour la réalisation d'applications internet résistantes à la charge et, dans le pire des cas, capables de gérer une dégradation "gracieuse" la qualité de service. La programmation événementielle est celle qui convient le mieux à la création de serveurs. En effet, elle offre une meilleure résistance à la charge que la programmation à base de threads. Cependant, en plus du problème des E/S asynchrones traité par Flash, il est très difficile avec un modèle de programmation événementiel d'ordonner les évènements en étant équitable et en ayant de bonnes performances. L'objectif de SEDA est de faciliter la conception d'applications web permettant des politiques de qualité de service. Dans ce cas, utiliser un modèle événementiel seul n'est pas la bonne solution. La programmation à base de threads n'est pas satisfaisante non plus du fait de son mauvais comportement à haute charge.

L'idée de SEDA est donc de se baser sur une programmation par étages. Le principe consiste à découper l'application en plusieurs étages. Chaque étage dispose d'une file de messages, qui lui sert à stocker les événement à traiter, d'un ensemble de threads, pour traiter ces événements, et d'un traitant qui effectue les opérations. Le traitant est l'opération qui est effectuée dans cet étage, comme par exemple vérifier si la page demandée est en cache pour un serveur de pages internet. Chaque événement à traiter est lu dans la file et affecté à un thread du groupe de threads. Une fois traité, un message est émis à destination de l'étage suivant (sauf dans le cas d'une étape terminale, comme une émission sur le réseau). La figure 1.5 illustre l'architecture de SEDA.

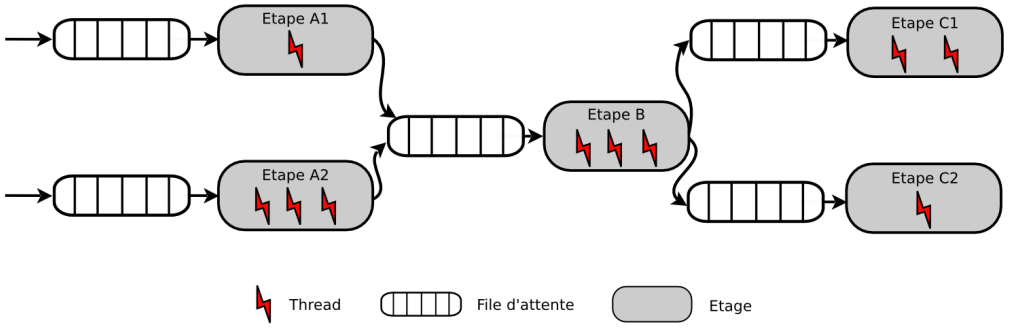

FIG. 1.5 – L'architecture de SEDA

Le problème du découpage de l'application est important. En effet, chaque étage dispose de ses propres ressources d'exécution. Un passage de message entre deux étages entraîne un changement de contexte pour que celui-ci soit traité dans l'étage suivant. Un découpage trop fin de l'application peut introduire de la latence à cause des coûts de communication entre les étages.

Deux optimisations ont été introduites. La première est la possibilité de faire varier dynamiquement le nombre de threads dans chaque étage pour mieux répondre à la demande. La seconde est le traitement des messages par lots, ce qui augmente le temps de réponse, puisque le système doit attendre d'avoir un certain nombre de messages avant de les traiter, mais qui augmente aussi le débit car les coûts de synchronisation et d'accès à la mémoire sont amortis.

Gestion de la qualité de service

Un autre aspect étudié par les créateurs de SEDA est la gestion de la qualité de service [25]. En effet, les contenus dynamiques sur les sites internet ont des coûts importants pour les systèmes sous-jacents. Actuellement la qualité de service est gérée par le système d'exploitation. Toutefois, cette situation n'est pas satisfaisante car le système n'a pas assez d'informations sur l'application. L'idée est d'utiliser SEDA et de lui ajouter des mécanismes de contrôle. En appliquant un contrôle sur chaque étage, il est possible de rejeter ou de ralentir les demandes qui consomment le plus de ressources. Par exemple, il peut être intéressant de mettre une priorité sur les demandes légères en terme de ressources.

Le contrôle dans SEDA s'effectue au moyen d'un mécanisme de jetons. Le jeton sert à autoriser l'ajout d'un message dans la file de l'étage. Le contrôleur délivre plus ou moins de jetons selon la politique en cours. Si aucun jeton n'est disponible, le message est rejeté.

Diverses améliorations sont possibles mais n'ont pas été implantées. Par exemple, lorsqu'un client est rejeté dans un étage, il a dû parcourir tous les étages précédents. Il y a donc à la fois un coût pour le système et pour le client. Il faudrait donc une communication entre les étages pour déceler au plus tôt cette situation et la traiter. Une autre amélioration pourrait être basée sur une coordination plus importante entre les étages. En effet, au lieu de confiner la gestion des ressources au niveau de chaque étage, les étages pourraient dialoguer entre eux pour prendre des décisions plus cohérentes. Ainsi un étage pourra réduire volontairement sa consommation en fonction des autres.

1.3.5 Click

Le projet Click [17] est un canevas modulaire et extensible pour la construction de routeurs. En effet, les routeurs actuels deviennent de plus en plus polyvalents. Ils disposent par exemple de fonctions annexes telles que le cryptage ou le décryptage de paquets, la possibilité de filtrer des paquets (pare-feu) ou encore la possibilité de créer des réseaux virtuels privés. Deux façons de concevoir les routeurs existent : soit un maximum d'opérations sont faites au niveau matériel, soit les opérations sont faites de façon logicielle. Dans le premier cas, l'avantage est la vitesse de traitement des informations mais le prix est généralement élevé, du fait de la complexité des circuits, et la flexibilité est très faible. En effet, la mise à jour d'une solution matérielle est problématique puisqu'il s'agit souvent de changer entièrement le matériel (pour par exemple ajouter de nouvelles fonctionnalités, ou encore corriger un problème ou une faille de sécurité). La solution logicielle parait donc mieux appropriée pour la conception de routeurs flexibles. Cependant cette solution se heurte souvent à des problèmes de performance. Le travail fait sur Click s'est donc découpé en deux phases. La première consistait proposer une solution n'utilisant les capacités que d'un seul processeur. La seconde phase était de proposer une évolution de la première solution permettant l'utilisation des machines multiprocesseurs. L'étude de Click qui suit est donc découpée selon ces deux axes.

Principes généraux

Tout comme SEDA, Click s'appuie sur une architecture à étages. Un programme Click est donc composé de multiples *éléments* reliés entre eux par des *connexions*. Cependant, contrairement à SEDA, l'interaction entre les éléments est effectuée de manière synchrone. Un élément est en fait une partie de l'application qui effectue un travail spécifique. Il peut effectuer tout type d'opération comme par exemple traiter les paquets en entrée, implémenter un compteur ou encore faire une opération de filtrage sur les paquets. Comme l'interaction est synchrone, la durée d'exécution des éléments n'est pas très importante. Les éléments sont interconnectés entre eux par un mécanisme de *ports*. Il y a deux types de ports : les ports d'entrées et ceux de sorties. Un port de sortie est connecté avec un port d'entrée et inversement. Les ports peuvent être connectés entre eux de deux manières : *push* et *pull*. Dans le premier cas, l'élément va envoyer les informations au prochain élément. C'est typiquement le cas d'un élément récupérant les arrivées sur la carte réseau et les insérant dans une file de paquets pour un traitement ultérieur. Une file est représentée par un élément prédéfini *Queue*. Une file possède un port d'entrée en mode *push*, et un port de sortie en mode *pull*. La figure 1.6 illustre une connexion entre trois éléments dont l'un est de type Queue. Le mode *pull* est utilisé lorsqu'un élément va aller chercher ses informations dans le ou les éléments précédents (comme par exemple dans une file de messages). Logiquement, il n'est possible de connecter entre eux

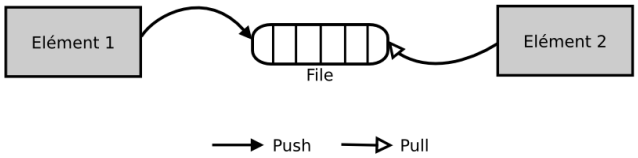

FIG. 1.6 – Un exemple de connexion d'élément avec Click

que des ports de même type. Il est possible de définir des ports génériques, appelés "agnostic ports" qui prennent la valeur des ports auxquels ils sont connectés. Pour leur exécution, les éléments sont placés dans une file d'attente et exécutés de façon périodique.

Click a été validé en réalisant un routeur IP. Pour montrer la modularité du système, un certain nombre d'extensions ont été réalisées. Par exemple des politiques de contrôle de flux ou encore de priorités ont pu être mises en place, "simplement" en rajoutant quelques éléments. Un mécanisme permettant de réinstaller à chaud une configuration aide à la mise en place de telles évolutions.

SMP Click

SMP Click [9] est une extension de Click visant à tirer parti des multiprocesseurs. Un des objectifs du projet est que le parallélisme de l'application doit être **transparent** pour les programmeurs de composants. Pour obtenir de bonnes performances, il faut faire très attention à deux sources coûts possibles : la gestion des caches et l'exclusion mutuelle des traitants.

La solution choisie est de créer un thread par processeur. Ce thread contient une liste d'éléments à exécuter périodiquement. Les éléments pouvant être insérés dans cette liste sont *PollDevice*, *ToDevice* et *PullToPush*. Lorsqu'un élément est exécuté, le chemin entier (jusqu'au push dans une file ou alors terminé par un ToDevice) est exécuté. Le modèle d'exécution est donc un modèle synchrone. Un élément PullToPush a été défini. Il est utilisé pour introduire un point de découplage au niveau de l'exécution du chemin si celui-ci s'avère coûteux. Ce point de découplage est inséré manuellement par le programmeur.

Deux options sont proposées pour répartir les éléments sur les ressources d'exécution. Il est possible d'effectuer la distribution soit de manière statique (en affectant statiquement les éléments aux différents processeurs) ou de manière dynamique. Dans ce dernier cas, les éléments sont répartis de manière à équilibrer la charge entre les processeurs. Périodiquement, le système échantillonne le nombre de cycles processeur nécessaires à l'exécution des différents éléments puis les répartit. Le coût de cet échantillonage est négligeable. Comme le montre Calarco *et al.* [8], ordonnancer les éléments de manière statique permet d'obtenir de meilleures performances. En effet, l'ordonnancement dynamique ne prend pas en compte le coût des défauts de cache alors que ceux-ci entraînent des réductions importantes de performances. Au contraire, l'ordonnancement statique peut se baser sur des données statistiques. Par exemple, il est très peu probable que FromDevice(x) et ToDevice(x) manipulent les mêmes données. Il n'est donc pas intéressant de les assigner au même processeur. À l'inverse, on peut statistiquement savoir que tel port du routeur dialogue plus souvent avec tel autre.

Différentes optimisations sont mises en place. Par exemple, les paquets sont traités par lots, ce qui réduit les coûts de synchronisation. En effet, pour manipuler les différentes listes,

il est nécessaire de mettre en place une exclusion mutuelle sur celle-ci. Le coût de l'exclusion mutuelle est important et ainsi il est intéressant de faire plusieurs opérations une fois le verrou possédé pour amortir ce coût. Pour ce qui est de l'exécution, le fait que les éléments à exécuter ne soient pas dans une liste globale permet d'éviter les coûts d'exclusion sur cette liste.

SMP Click a été testé sur deux configurations différentes, le routeur IP de Click et un routeur implémentant un VPN[5]. Dans le premier cas, SMP Click n'est pas capable de tirer profit de plus de deux processeurs. Cela s'explique par le fait que peu d'opérations coûteuses sont effectuées en parallèle. C'est dans le second cas que SMP Click va permettre d'obtenir de bonnes performances. En effet, les coûteuses opérations de cryptage et de décryptage peuvent être effectuées en parallèle.

SMP Click permet donc de construire de façon modulaire des routeurs ayant des performances comparables voir supérieures dans certains cas à des routeurs commerciaux. Toutefois, pour arriver à tirer profit des multiprocesseurs, il faut que le routeur fasse des opérations complexes comme le cryptage/décryptage. Un des inconvénients de SMP Click est que l'insertion de points de découplage (PullToPush) est à la charge du programmeur et ne peut pas évoluer en fonction, par exemple, de la charge du routeur. Il n'existe pas de manière automatique d'insérer ces point de découplage. Cette partie peut donc être complexe pour le programmeur.

1.4 Impact du support d'exécution

Dans certains cas de figure, l'architecture sous-jacente du noyau et notamment la façon dont ce dernier gère les threads peut aussi influencer le mode de programmation à choisir. Dans cette partie nous expliquons dans un premier temps les choix qui peuvent être faits au niveau du système d'exploitation. Ensuite nous présentons les exemples de Mach et de Fluke. Enfin la solution de l'ordonnancement coopératif est expliquée.

1.4.1 Modèle de processus et modèle d'interruption

Au niveau du système d'exploitation, deux modèles différents peuvent être choisis : soit un modèle basé sur les processus, soit un modèle basé sur les interruptions. La différence principale entre ces deux modèles vient de l'endroit où est stocké l'état du thread. Dans le modèle basé sur les processus, chaque thread dispose de sa propre pile. Ainsi lors d'une interruption, par exemple, l'état du thread est sauvegardé dans cette pile. C'est la méthode qui est choisie dans la plupart des systèmes d'exploitation. Dans le cas du modèle basé sur les interruptions, il n'y a qu'une seule pile par processeur qui connaît l'état du processus courant. Lors de l'interruption, le thread est chargé de sauvegarder son état avant de s'endormir. Ces sauvegardes explicites sont appelées "continuations" [11]. La différence entre les deux modèles provient essentiellement du fait que dans un cas, c'est le système qui à la charge de sauvegarder l'état d'un processus dans sa pile. Dans un autre cas c'est au programmeur de le faire.

L'avantage majeur du modèle basé sur les interruptions est sa consommation mémoire. En effet, celui-ci n'utilise qu'une seule et unique pile (et non pas une pile par thread). En contrepartie, cela implique que le programmeur s'acquitte manuellement de la gestion de la mémoire. De plus avec le modèle basé sur les interruptions, le thread en cours ne peut pas être complètement préemptible, c'est à dire qu'il ne peut pas être interrompu par le noyau à tout moment puisqu'il doit sauvegarder son état préalablement.

[5]Virtual Private Network, réseau privé virtuel

Mach

Mach [11] est un micro-noyau qui, dans ses premières versions, était basé sur un modèle à base de processus. Cependant les auteurs de Mach se sont rendus compte que le mélange des deux solutions pouvait être avantageux pour le système, puisqu'un tel procédé peut augmenter à la fois les performances et diminuer la quantité de mémoire utilisée. Un processus qui bloque va pouvoir utiliser au choix l'un ou l'autre des modèles. S'il choisit le modèle basé sur les interruptions, il va devoir sauvegarder les informations importantes de sa pile puis libérer cette dernière. Lorsque le thread est réveillé, il restaure son état. Cette gestion de la consommation de la mémoire peut être utile dans Mach. En effet, Mach a été conçu pour supporter un grand nombre d'architectures et notamment les systèmes embarqués. Dans ce cas là, une pile de 4Ko par thread peut être trop importante pour le système. Cependant, dans certains cas, il peut être intéressant d'utiliser le modèle de processus, comme par exemple dans le cas d'un défaut de page en espace noyau. En effet, ce cas là est difficile à traiter étant donné que le thread peut se bloquer à n'importe quel moment dans le noyau. La mise en place du modèle basé sur les interruptions s'est faite en recherchant les parties des fonctions du noyau qui pouvaient bloquer, et en coupant la fonction autour de ce point de blocage. De plus, l'utilisation de continuations va pouvoir améliorer la communication entre processus. En effet, Mach dispose d'un mécanisme de libération de pile (*stack handoff*) avec lequel le thread réveillé va pouvoir utiliser la pile du thread précédent. Par exemple, dans le cas d'un dialogue entre un client et un serveur, un client émet une requête, entre en espace noyau et sauve son état. Le noyau cherche un serveur disponible, suspend le thread courant et réveille le nouveau thread. Le serveur restaure son état grâce aux continuations, récupère le message, grâce à l'utilisation de la pile du thread précédent, et fait ce qu'il a à faire. La communication entre le client et le serveur a pu être effectuée simplement.

Fluke

Le noyau Fluke [12] a été crée pour effectuer des tests et étudier les différences entre les deux modèles présentés précédemment. Tout comme Mach, Fluke est capable d'utiliser les deux modèles. Fluke dispose d'une API atomique et chaque appel système est interruptible et redémarrable. Cette API permet à un thread d'accéder à l'état complet d'un autre thread. Le fait que les opérations soit interruptibles et redémarrables est une propriété essentielle. Il existe différents types d'applications, les "courtes" qui ne se bloquent pas et ne sont pas interruptibles et les "longues" qui peuvent potentiellement être interrompues et redémarrées.

Le modèle choisi joue directement sur la conception de l'API[6]. En effet, un modèle basé sur les interruptions aura tendance à plutôt avoir des appels systèmes courts et atomiques alors que l'autre pourra proposer des appels plus longs. En plus d'être complètement redémarrable et interruptible, une API atomique doit garantir deux propriétés : la vivacité et la correction. La vivacité est le fait de garantir qu'un thread peut accéder à l'état d'un autre thread sans attendre que son opération soit finie. La correction est la propriété énonçant que si un thread est interrompu et redémarré, il l'est dans le même état que précédemment. Pour garantir l'atomicité, si un thread demande l'état d'un thread en cours, il est nécessaire que cette opération soit annulée ou soit finie avant qu'un autre thread puisse accéder à son état. Cela dépend de l'opération qu'il est en train d'effectuer à cause de la propriété de vivacité.

[6] *Application Programming Interface*, Interface de Programmation d'Application

Comme Fluke a été conçu pour supporter les deux types de modèles, un de ses intérêts est de pouvoir les comparer selon différents critères. Nous pouvons citer notamment :

- L'API exportée : identique dans les deux cas. La solution choisie dans Fluke est de masquer les continuations au code utilisateur.
- Préemption : Un modèle basé sur les processus peut implémenter tout type de préemption alors qu'un modèle basé sur les interruptions ne peut implémenter qu'une préemption partielle, ce qui a des conséquences sur la latence.
- Performances : Les performances entre ces deux solutions sont quasiment équivalentes.
- Utilisation mémoire : Sans surprise, l'utilisation mémoire est meilleure dans le cas du modèle basé sur les interruptions.
- Le support du matériel pour un modèle : Certains matériels tels que l'architecture x86 supposent que chaque thread dispose d'une pile. La pile est gérée automatiquement par le processeur. Il y a donc un coût supplémentaire pour utiliser un modèle basé sur les interruptions.

1.4.2 L'ordonnancement coopératif

L'idée de l'ordonnancement coopératif est de corriger les problèmes soulevés par l'utilisation de threads décrits dans le partie 1.1.2. Les problèmes sont essentiellement liés à l'ordonnancement de l'exécution plutôt qu'à la programmation par threads elle même. En effet, un thread qui ne serait pas préempté n'aurait pas de problème de synchronisation. L'idée est donc d'assurer qu'un thread ne redonne la main au système qu'au moment qu'il juge opportun. Ainsi un thread n'est pas préempté dans une section critique ce qui évite tous les problèmes de synchronisation. Ceci n'est valable que dans l'hypothèse où l'application n'utilise qu'un seul processeur.

L'idée de Adya *et al.* [5] est se baser sur l'ordonnancement coopératif pour proposer une solution permettant de combiner la programmation événementielle et la programmation à base de threads. L'idée est d'arriver à prendre le meilleur des deux mondes et de pouvoir contenter les utilisateurs. Deux principaux concepts sont distingués, le concept de *gestion de pile* et celui de *gestion de tâches*. La gestion de taches est la façon dont sont entrelacées les exécutions des différents processus du système. Trois différentes manières d'ordonnancer les processus existent.

- Soit le processus peut être interrompu à n'importe quel instant. Ce mode est appelé *préemption*. Cela permet d'avoir un parallélisme réel entre les différents threads.
- Soit le processus s'exécute en entier sans jamais laisser la main à un autre processus. Ce mode est appelé *run-to-completion*. Il n'y a pas de parallélisme entre les différents threads dans ce mode.
- Soit le processus laisse la main à des moments bien définis. Il s'agit de l'ordonnancement coopératif. C'est une solution intermédiaire qui apporte un pseudo parallélisme tout en minimisant les risques entraînés par la préemption.

Les manières de gérer la pile sont décrites dans la partie 1.4.1. Une gestion de pile automatique correspond à un programme multi-threadé. En effet, le code de l'application va être en un bloc interruptible et redémarrable. Une gestion de la pile manuelle va correspondre à un programme événementiel. Dans ce cas, le programmeur doit définir des méthodes à exécuter lors d'un événement et quelles sont les informations à sauvegarder lors de la fin d'une méthode.

L'idée est donc de pouvoir mélanger les modes de programmation événementielle et multi-

threadée pour simplifier la programmation et contenter les programmeurs. En effet, il est parfois plus judicieux d'utiliser le modèle de gestion automatique de la pile. Un exemple concerne les fonctions qui ne font pas d'E/S. Le problème de l'utilisation de la programmation événementielle est le découpage du code de l'application. En effet, plus il y a d'E/S à l'intérieur d'une fonction, plus cette dernière va être découpée ce qui réduit la lisibilité générale du code. Le déboguage ainsi que l'évolution du code en sont aussi affectés. En effet, si une fonction évolue et se met à faire des E/S, toutes les fonctions liées risquent d'être découpées. Cependant l'utilisation de la programmation événementielle limite le nombre de processus en cours d'exécution et permet ainsi une meilleur montée en charge.

Pour pouvoir utiliser les deux méthodes, la solution proposée se base sur des "fibres" (*fibers*) qui sont des threads ordonnancés de façon coopérative. Il ne peut y avoir qu'une seule fibre s'exécutant à la fois à un instant donné. L'ordonnanceur se déroule dans une fibre principale et ordonnance à la fois du code utilisant une pile gérée de façon automatique et du code utilisant une pile gérée de façon manuelle. Dès que le code en cours d'exécution veut effectuer une opération bloquante, il est démarré dans une nouvelle fibre. Cette fibre va se bloquer, par exemple en attente d'une E/S. La fibre se bloque et laisse la main à la fibre principale. La gestion du blocage dépend du type de gestion de la pile choisie. La figure 1.7 illustre un exemple de blocage sur une E/S. Les fibres sont un mécanisme disponible

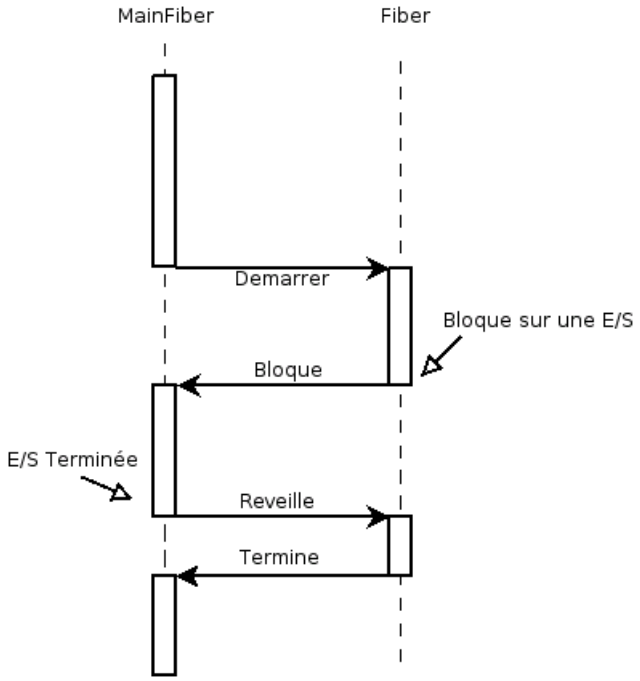

FIG. 1.7 – Un exemple de connexion de gestion d'E/S avec les fibres

dans les systèmes d'exploitation Windows. Cependant des solutions équivalentes peuvent être appliquées à d'autres systèmes.

Cette solution permet de combiner les avantages des deux types de programmation tout en arrivant à contenter les programmeurs habitués à un type de programmation en particulier. Cependant, étant donné qu'il n'y a qu'une seule fibre active à un moment donné, l'ordonnancement coopératif ne permet pas une utilisation efficace des multiprocesseurs.

1.4.3 Capriccio

Capriccio [24, 23] est un projet visant à proposer une bibliothèque de threads offrant de bonnes performances pour les serveurs à haute concurrence. Capriccio montre qu'il est tout à fait envisageable d'utiliser la programmation par threads pour construire des serveurs robustes. Cette bibliothèque a pour objectif de proposer de meilleures performances que les autres solutions et permet aussi de gérer de nombreux threads dans le système avec une dégradation de performances réduite.

Deux solutions étaient possibles : soit utiliser des threads de niveau noyau, soit utiliser des threads de niveau utilisateur. Les concepteurs de Capriccio ont opté pour la deuxième solution. Les raisons qui ont motivé ce choix sont notamment la légèreté des threads de niveau utilisateur ainsi que leur flexibilité. En effet, la bibliothèque peut être aisément adaptée pour prendre en compte les évolutions du noyau en termes d'interface et de fonctionnalités (par exemple l'introduction d'E/S asynchrones). Il est aussi possible d'utiliser un ordonnanceur plus adapté à l'application que celui plus général du noyau. Enfin, manipuler des threads de niveau utilisateur évite de coûteux passages en mode noyau. La contrepartie est de devoir intercepter les appels systèmes bloquants pour les transformer en appels non bloquants afin que toute l'application ne soit pas bloquée, ce qui rajoute une couche dans les appels. L'ordonnancement utilisé est de type coopératif (voir section 1.4.2). Un des intérêts principaux est d'enlever les problèmes de synchronisation sur les machines mono-processeur. L'inconvénient est que ce système ne permet pas de tirer parti des machines disposant de plusieurs processeurs. Les futures versions essaieront de proposer une solution à ce problème.

Un autre problème à résoudre est celui de la consommation mémoire due à l'utilisation de nombreux threads. En effet, généralement la taille de la pile d'un thread est fixe. La plupart des threads n'utilisent pas beaucoup cette pile et une grande quantité d'espace mémoire est ainsi gaspillée. Pour résoudre ce problème, Capriccio introduit un mécanisme de pile dynamique. Dans un premier temps le compilateur va créer le graphe représentant les différents chemins que peut suivre l'exécution d'un programme. Les noeuds du graphe sont les procédures, décorées par l'espace mémoire nécessaire pour sauvegarder son contexte. Il est donc possible de déterminer la taille maximum de la pile lors de la compilation, si le graphe est sans boucle (s'il n'y a pas de fonctions récursives), en prenant la taille du plus grand chemin. Une première estimation de la taille de la pile peut donc être fixée à la compilation. Il est cependant nécessaire que la taille de cette pile puisse évoluer. Par exemple, le nombre d'appels récursifs d'une fonction n'est pas prévisible et la taille de la pile ne l'est donc pas non plus. De même, il est dommage de réserver la taille du plus grand chemin si celui-ci n'est pas parcouru. Pour cela, des points de contrôle sont insérés. Ces points de contrôle vont vérifier que l'espace mémoire réservé est compris dans un certain intervalle. Si ce n'est pas le cas, il va être possible d'augmenter ou de réduire la taille de la pile.

Une autre optimisation présente dans Capriccio est l'ordonnancement en fonction des ressources restantes du système. En effet, il peut être intéressant de choisir une connexion qui se termine pour libérer ses ressources plutôt qu'une connexion en cours. Pour cela, Capriccio se base sur le graphe des appels bloquants. Ce graphe représente tous les appels bloquants du système et les chemins entre eux. Les noeuds ainsi que les arcs sont pondérés par le temps nécessaire. Un thread bloqué le sera donc forcément dans un de ces états. Ce graphe est généré dynamiquement durant l'exécution. Il va donc être possible d'identifier les appels dont le temps augmente en raison du manque de ressources et d'ordonnancer par exemple des threads qui libèrent ces ressources. Un autre avantage de ce graphe est de pouvoir déterminer quel

sont les opérations qui prennent du temps (durée d'un arc). Les avantages de l'ordonnancement coopératif proviennent du fait que les threads relâchent eux-mêmes la main. Il est donc nécessaire pour la réactivité que l'opération ne soit pas trop longue. Grâce à ce système, le programmeur peut facilement remarquer les fonctions qui bénéficieraient d'un découpage plus fin.

1.5 Les approches basées sur le langage

Dans les parties précédentes, nous avons essentiellement vu des approches modifiant la façon d'ordonnancer les opérations. Des approches de plus haut niveau sont possibles. L'idée est de séparer la programmation du support d'exécution, en proposant un langage adapté à la programmation parallèle. Dans cette partie, nous présenterons deux approches basées sur le langage qui essaient de proposer une alternative aux méthodes de programmation traditionnelles.

1.5.1 Une approche basée sur le langage Haskell

Pend *et al.* [21] décrivent une manière d'unir la programmation événementielle et la programmation par threads. En effet, pour les raisons décrites précédemment, il peut être intéressant d'utiliser tantôt l'une et tantôt l'autre. Les auteurs partent du principe que la programmation par threads et la programmation par évènements sont deux méthodes équivalentes, comme montré par Lauer *et al.* [14]. Par exemple, l'ordonnanceur présent dans un système basé sur les threads peut être assimilé à la boucle de contrôle présente lors de l'utilisation d'événements. Une des idées directrices est que, si ces deux ordonnanceurs sont similaires, celui utilisé avec les événements est bien plus flexible que celui utilisé avec les threads. En effet, un des problèmes de la programmation par threads est de masquer l'ordonnanceur et donc que ce dernier soit très peu flexible, puisque le programmeur ne peut pas avoir de contrôle dessus. A l'inverse, avec les évènements, il est possible de rajouter des traitements pour lesquels le système n'était pas prévu.

La solution est basée sur le langage Haskell qui est de la famille des langages ML et est donc un langage fonctionnel. L'idée est de proposer une bibliothèque permettant d'avoir des threads de niveau "application". Les threads de niveau application se démarquent des habituels threads de niveau utilisateur par le fait que l'ordonnanceur de threads fait partie de l'application. Ainsi le parallèle avec la programmation par événements est encore plus évident. Il va donc être possible de faire évoluer l'ordonnanceur pour lui ajouter de nouvelles fonctions. Cet ordonnanceur va à la fois traiter des threads et des événements, ce qui permet d'utiliser l'un ou l'autre de ces deux styles de programmation au sein de l'application. Pour les threads, les événements sont représentés par les appels système. Un thread va être exécuté jusqu'au moment où il relâche explicitement la main, ou lorsqu'il fait un appel système. Cette approche est similaire à l'ordonnancement coopératif.

Pour pouvoir exploiter les capacités des multiprocesseurs, plusieurs boucles d'événements peuvent être lancées simultanément. Ces boucles vont récupérer des événements à traiter dans une file. Il est possible d'utiliser les primitives d'exclusion mutuelle existantes ou d'en ajouter de nouvelles dans l'ordonnanceur.

Cette approche permet d'unifier la programmation événementielle et la programmation par thread en essayant de tirer parti des avantages de chacun. Cependant l'inconvénient principal est que ce mécanisme n'est disponible que dans le langage Haskell. En effet, cette solution tire

parti des possibilités proposées par les langages fonctionnels. La transposer dans un langage impératif serait non seulement difficile, car la solution utilise la plupart des subtilités du langage Haskell, mais en plus, comme le soulignent ses auteurs, le résultat serait lui aussi difficile à utiliser dans un autre langage.

1.5.2 Flux

Flux [7] est un langage qui a été imaginé pour permettre le développement rapide et sûr de serveurs à hautes performances. L'objectif de Flux est de fournir une abstraction de haut niveau décrivant le comportement de l'application et un compilateur/générateur de code permettant d'obtenir un code fiable et performant. Un langage de haut niveau est utilisé pour masquer le support d'exécution sous-jacent au programmeur. Ainsi, il va être possible d'utiliser un modèle basé sur les événements ou un modèle basé sur les threads de manière indépendante au développement de l'application.

Flux étant un langage, il dispose de sa propre syntaxe. l'exécution peut être vue comme un flux sans cycles. La figure 1.8, inspirée de celle présentée par l'auteur [7], représente schématiquement une exécution.

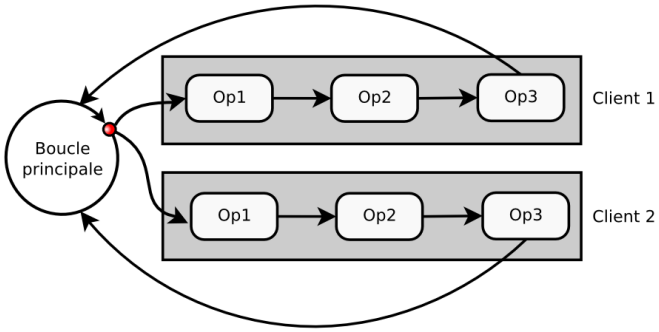

FIG. 1.8 – Un exemple d'exécution avec Flux

La syntaxe de Flux permet :

- de décrire les différentes opérations, via des noeuds concrets. Par exemple, on va avoir : `OP1 (int entrée) => (int sortie)` si l'opération 1 est une opération qui prend un entier en entrée et retourne un entier.
- de décrire les liens entre les opérations, via des noeuds abstraits. Dans l'exemple précèdent on aura : `NoeudLien = OP1 -> OP2 -> OP3`.
- de décrire des transitions conditionnelles.
- de permettre la levée et la récupération d'erreurs.
- de définir des contraintes pour garantir l'atomicité de certaines opérations. Par exemple, `atomic OP1 :{x,y}` va indiquer au système de prendre un verrou sur x et y avant d'exécuter OP1.

L'atomicité est un point clé de Flux. Le programmeur peut définir des contraintes d'exclusion mutuelle sur les opérations. Le schéma de synchronisation adopté est de type lecteur/-rédacteur, c'est à dire que plusieurs processus peuvent effectuer l'opération en même temps mais que l'écriture ne peut être faite qu'en exclusion mutuelle. Il est possible de simuler un comportement pessimiste en déclarant toute prise de verrou comme une écriture. L'intérêt

de définir les contraintes d'intégrité à ce niveau là est que le code généré est sans possibilité d'interblocage. Ainsi, un des points reprochés à la programmation par threads est levé. Par ailleurs, il est possible de générer du code plus efficace, puisque selon le modèle d'exécution choisi, il peut ne pas être nécessaire d'appliquer des méthodes d'exclusion mutuelle, comme par exemple dans le cas d'une exécution de type événementielle sur un seul processeur. Pour garantir le fait que le programme soit sans interblocages, le système peut générer un code qui acquiert les verrous toujours dans le même ordre. Cet ordre est basé sur l'ordre alphabétique. Combiné au fait que le programme soit sans cycle, et donc qu'il ne peut pas demander un verrou qu'il possède, cela permet d'éliminer la possibilité d'avoir des situations d'interblocages.

Il est possible d'utiliser différents supports d'exécution tels que l'allocation d'un thread par connexion sans borne sur le nombre de threads ou avec une borne, ou encore une exécution de type événementielle en simulant les E/S asynchrones à la manière de Flash (voir section 1.3.1). Enfin, d'autres modèles peuvent être utilisés tels que celui proposé par SEDA, qui est en cours de réalisation. De ce fait, les évaluations de cette dernière solution ne sont pas disponibles. Flux dispose enfin d'un mécanisme permettant, via l'analyse des traces d'exécution, le détection des goulots d'étranglement du système.

1.6 Autres solutions

Dans cette partie, les autres solutions qui ont été proposées pour permettre de tirer parti des multiprocesseurs seront présentés. Nous verrons notamment les cas d'OpenMP et de MPI.

1.6.1 OpenMP

OpenMP [3] est un projet ayant pour but de permettre aux programmeur de réaliser plus aisément des applications tirant parti de la puissance des multiprocesseurs. La stratégie choisie est de proposer des directives de compilation simplifiant la programmation multi-threadée. Grâce à ces directives, il est possible de déclarer quelles sont les sections qui peuvent être parallélisées, quelles sont les sections à exécuter en exclusion mutuelle ou encore de définir des variables privées ou partagées pour les différents threads. OpenMP est basé sur un modèle de type “fork-join”, c'est-à-dire que N processus vont être crées, vont s'exécuter et vont faire un rendez-vous avant de continuer l'exécution du code dans le thread principal.

Un des intérêts d'OpenMP est de simplifier la programmation parallèle en aidant le programmeur à paralléliser son application. Il est, en effet, possible de paralléliser cette dernière de manière incrémentale. Il est aussi possible de paramétrer le nombre de threads utilisés grâce aux variables d'environnement.

Cependant, l'aide qu'apporte OpenMP est plutôt d'ordre syntaxique. En effet, les problèmes de synchronisation pour garantir la cohérence des variables partagées sont toujours présents même s'ils sont simplifiés grâce à certaines primitives. Le coût, en terme de temps d'exécution, engendré par ce type de synchronisation est aussi toujours présent. Enfin, le coût en terme de temps de développement, est lui augmenté par la nécessité de se familiariser avec de nouvelles primitives.

1.6.2 MPI

MPI [2] est un standard dans l'industrie des grappes de calcul. Il peut être vu comme le pendant d'OpenMP pour les systèmes à mémoire distribuée. Un système à mémoire distribuée

est par exemple une grappe d'ordinateurs. Pour que les processus communiquent, il n'est plus possible comme dans OpenMP d'utiliser la mémoire partagée. MPI se base donc sur un système de passage de messages. De la même façon qu'avec OpenMP, il est possible, par exemple, d'utiliser des méthodes de synchronisation entre les processus, telles que des barrières. Il va aussi être possible d'utiliser des primitives non bloquantes (asynchrones) pour permettre le recouvrement des coûts de communication.

MPI peut être vu comme un modèle à la fois événementiel (car il réagit à la réception de messages) et à la fois multi-processus (puisque plusieurs processus effectuent des opérations). Les communications entre les processus sont à la charge de l'utilisateur. Comme il s'agit d'un modèle à base de messages et non pas de mémoire partagée, il n'y a pas de problème d'accès concurrents aux données. Par contre, un coût est induit par cette utilisation de messages.

MPI est donc une solution intéressante pour les serveurs de calcul à haute performance, dont l'exécution peut être parallélisée de manière relativement indépendante. Cependant, pour les serveurs avec des données partagées ou des serveurs ayants des couplages forts, cette solution n'est pas forcement satisfaisante du fait du coût engendré par l'utilisation d'une communication par messages.

1.7 Synthèse

Ce chapitre avait pour objectif de présenter les différentes techniques existantes permettant de programmer des applications performantes. Les deux méthodes généralement utilisées sont la programmation par threads et la programmation événementielle.

Le modèle de programmation à base de threads est probablement le plus utilisé à l'heure actuelle. Il est assez simple dans le sens où elle conserve l'aspect séquentiel de la programmation traditionnelle. Il permet aussi une bonne utilisations des multiprocesseurs. Cependant, ce type de programmation se heurte à des problèmes de performance lorsque le nombre de threads augmente dans le système. De plus, la synchronisation est difficile à cause du comportement non déterministe de l'ordonnanceur. Enfin, la portabilité de l'application dépend de la présence du support des threads de niveau noyau dans le système d'exploitation ou de la présence d'une bibliothèque permettant d'utiliser des threads de niveau utilisateur.

La programmation événementielle peut résoudre plusieurs de ces problèmes. Ceci est dû au fait qu'un processus n'est pas suspendu au bénéfice d'un autre processus. De ce fait, les problèmes de concurrence et de passage à l'échelle sont résolus. Comme ce type de programmation n'est pas (ou peu) dépendant d'une bibliothèque, la portabilité est assurée entre les différents systèmes d'exploitation. Cependant le fait que le programme soit découpé en de multiple traitants réduit la lisibilité générale du code, ce qui réduit aussi la facilité de programmation. Un autre problème est la présence d'E/S asynchrones sur le système. Si de telles E/S ne sont pas présentes, la performance de la programmation par événements peut fortement baisser. Enfin, étant donné qu'il n'y a qu'un seul processus, cette méthode ne permet pas de tirer parti de la présence de plusieurs processeurs.

Les inconvénients de la programmation à base de threads et de la programmation événementielle ont incité les chercheurs à trouver des solutions pour concevoir des programmes plus efficaces, plus sûrs et faciliter la tâche des programmeurs. Certains projets ont cherché à améliorer la programmation par threads. C'est le cas notamment de Capriccio [24]. En se basant sur un nouveau type d'ordonnancement et en fournissant une bibliothèque de threads plus performante, Capriccio propose une méthode pour construire des serveurs à hautes per-

formances. Toutefois, Capriccio ne permet pas actuellement de bénéficier des possibilités des multiprocesseurs et en ajouter le support réintroduirait les problèmes d'accès concurrents à la mémoire. Dans le cas de services Web, cette solution pourrait être envisagée, les requêtes étant généralement indépendantes. Cependant, ce n'est pas le cas de la plupart des applications.

D'autres projets se sont intéressés à l'amélioration de la programmation événementielle. C'est le cas de Flash [19] et de la bibliothèque Libasync [27]. Flash propose une solution pour effectuer les E/S bloquantes alors que Libasync est un bibliothèque pour décharger le programmeur de certaines tâches (gestion de la mémoire, gestion de la boucle de contrôle, ...) et tirer parti des architecture parallèles. Cependant, le support des multiprocesseurs ne peut être fait qu'à gros grain (de l'ordre du traitant d'événements). Dans certains cas, garder un verrou pendant la durée complète d'un traitant peut être trop coûteux pour le système et ne permet pas d'obtenir un parallélisme suffisant pour atteindre de bonnes performances. Un autre inconvénient est que l'éclatement du programme en de multiples traitants est toujours présent, bien que sa gestion soit simplifiée.

D'autres projets se sont intéressés aux architectures logicielles hybrides, c'est à dire qui mélangent une exécution de type événementielle avec l'utilisation de plusieurs threads. Il s'agit des projets SEDA [26], StagedServer [13] et Click [17]. SEDA et StagedServer sont tous deux basés sur des architectures à étages. Le programme est composé d'étages communiquant de manière asynchrone (événementielle) grâce à des files de messages. Chaque étage dispose de ses propres ressources d'exécution dont la quantité peut être adaptable. Ainsi, cette utilisation des étages marie des interactions événementielles, entre les étages, avec une exécution multi-threadée — chaque étage disposant de son lot de threads. Les problèmes de synchronisation deviennent, quant à eux, confinés au niveau de chaque étage, ce qui en simplifie la gestion.

SMP-Click [9] à été motivé par l'objectif de construire des routeurs modulaires et performants pour des machines multiprocesseurs. L'application est composés d'éléments dont la durée d'exécution peut être parfois très courte. Dans le cas de SMP-Click, l'exécution est entièrement synchrone mais il est possible d'insérer des points de découplage si le chemin s'avère trop coûteux. Deux problèmes surviennent toutefois dans l'architecture de Click :

- La mise en place des points de découplage est faite manuellement par le programmeur de l'application. Cette mise en place ne peut donc résulter que de tests approfondis sur les différentes possibilités.
- La mise en place de ces points de découplage est statique. En effet, pour cela, le programmeur doit insérer des éléments *PullToPush* directement dans son application. Ainsi, Click ne permet pas de faire évoluer le comportement de l'application en cours d'exécution. Cela suppose donc que la charge présente sur le programme varie peut au cours du temps et reste proche de celle pour laquelle l'application a été conçue.

D'autres projets abordent le problème de la programmation parallèle avec une approche langage, en proposant soit des extensions des langages existants, soit de nouveaux langages. Toutefois cette solution, si elle améliore le confort de programmation, n'améliore pas le fonctionnement des environnements d'exécution sous-jacents.

Enfin, pour conclure cette synthèse, notons que les différentes solutions étudiées satisfont toutes des objectifs variés, tels que la simplicité d'utilisation, l'exploitation de tous les processeurs présents au sein d'une machine, la résistance à la montée en charge, ou encore la fourniture de services annexes tels que la gestion de la dégradation de service. Le tableau 1.9 présente une synthèse des différentes solutions étudiées en fonction de ces critères.

Type	**Simplicité**	**Gestion concurrence**	**Passage à l'échelle**	**Support multiprocesseurs**	**Portabilité**	**Autres services (dégradation, ...)**
Threads	***	*	*	***	*	*
Évènements	*	***	***	*	**	*
Libasync	**	**	***	**	**	*
Flash	*	***	*	*	***	*
SEDA	***	**	***	***	**	***
Click	***	**	***	***	*	**
Coopératif	**	***	***	*	**	*
Capriccio	***	***	***	*	**	**

FIG. 1.9 – Tableau de synthèse de l'état de l'art

Chapitre 2

Contribution

Dans le chapitre précédent, nous avons dressé un état de l'art des techniques permettant de programmer des applications performantes. Dans ce chapitre, nous présentons notre contribution qui consiste à améliorer le modèle de programmation par étages afin de le rendre ce modèle d'exécution dynamiquement adaptable.

2.1 Vers des environnements d'exécution adaptables

Nous pensons que la programmation à base d'étages est une bonne solution pour le développement de programmes robustes. Cependant, nous pensons qu'une conception à base d'étages gagnerait à être plus flexible. En effet, comme expliqué précédemment, la communication entre les différents étages est faite de manière asynchrone (événementielle). Nous pensons que dans certains cas, il peut être intéressant que cette communication soit faite de manière synchrone. Par exemple, dans le cas d'un serveur Web, un modèle à base de threads (exécution synchrone) permet d'obtenir un bon temps de réponse lorsque le débit de requêtes entrantes est faible. Par contre, ce modèle ne permet pas un passage à l'échelle efficace. Un modèle à base d'étages (exécution événementielle) permet lui d'obtenir de bons débits, notamment lorsque le nombre de requêtes entrantes est élevé.

Notre idée est donc de séparer le programme de son modèle d'exécution sous-jacent. L'utilisation conjointe des modèles d'exécution synchrones et asynchrone est déjà présente dans Click. Toutefois, nous voulons que le modèle d'exécution soit reconfigurable dynamiquement en fonction de différents paramètres et surtout que le modèle d'exécution soit transparent pour le programmeur. Ce n'est pas le cas dans Click, puisque le programmeur doit manuellement insérer des points de découplage et que ces points ne peuvent pas évoluer dynamiquement. Flux a également essayé d'exploiter cette voie, en séparant le code du modèle d'exécution. La solution choisie a été de proposer un nouveau langage et de générer le code correspondant pour différents modèles d'exécution (threads ou évènements). Cependant, il n'est pas possible dans Flux de passer d'un mode d'exécution à un autre dynamiquement.

Un autre intérêt de découpler l'application de l'environnement d'exécution est que cela permettra d'adapter automatiquement le programme à l'architecture matérielle sous-jacente. Avec l'évolution des processeurs et notamment du nombre d'unités d'exécution, il est intéressant pour un programme de pouvoir s'adapter automatiquement à une nouvelle architecture matérielle. Prenons l'exemple d'une application développée pour un serveur. Si cette application est couplée au nombre de processeurs disponibles, alors lors d'une mise à jour du matériel,

l'application ne pourra plus fonctionner de manière optimale. Le découpage synchrone/asynchrone va permettre d'obtenir la structuration de base pour lier les composants aux ressources d'exécution. La figure 2.1 donne un exemple de répartition des blocs de composants sur les ressources d'exécution. L'évolution de ce de ce découpage permettra d'adapter dynamiquement le programme à l'architecture sous-jacente.

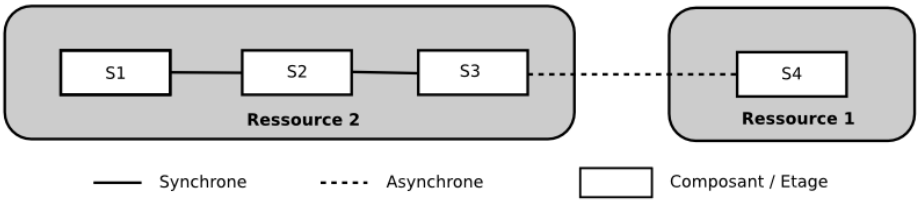

FIG. 2.1 – Un exemple d'affectation des composants aux ressources d'exécution

Notre objectif est donc de proposer une nouvelle architecture qui permette de changer dynamiquement de mode d'exécution en fonction de différents paramètres, tels que les ressources disponibles ou encore la charge courante. L'exécution pourra se faire soit de manière totalement asynchrone (équivalente au modèle à étages), soit de manière totalement synchrone (équivalent à un modèle à base de threads), soit de manière partiellement synchrone (équivalent à ce qui est fait dans Click).

Pour les travaux présentés dans ce rapport, nous avons réalisé cette architecture en nous basant sur une implantation en Java de SEDA, appelée Sandstorm. Nous présentons, dans la suite du rapport, les modifications que nous avons du faire à Sandstorm afin de permettre son changement dynamique de mode d'exécution. Cette première implémentation a pour but de permettre de déterminer qu'elles sont les heuristiques importantes pour adapter dynamiquement le modèle d'exécution. Notre objectif, à terme, est d'intégrer ces travaux au modèle de composants Fractal, et notamment à son implémentation en langage C, appelée Think. En effet, la programmation à base de composants correspond parfaitement au principe de découpage en étages. Tout comme les étages, les composants sont indépendants et communiquent via des interfaces. Habituellement, l'exécution de composants Think est faite de manière synchrone, mais il a récemment été introduit la possibilité d'exécuter les composants de façon asynchrone [16].

2.2 Architecture de notre proposition

Dans cette partie, les différents modèles d'exécution possibles pour une application seront présentés. Nous proposerons ensuite une évolution de l'architecture à base d'étages permettant la reconfiguration automatique du modèle d'exécution lorsque cela est nécessaire.

2.2.1 Modèles d'exécution

Comme il a été expliqué dans la partie précédente, un des objectifs du projet est de pouvoir faire varier dynamiquement le modèle d'exécution. La figure 2.2 illustre les différents modèles d'exécution possibles. Trois modèles différents sont présentés :

– (a) *représente une exécution totalement asynchrone (événementielle).* Cette exécution est similaire à celle utilisée dans les architectures à étage classiques. L'étage S1 va

mettre un message dans la file de l'étage S2 puis continuer son exécution. L'étage S2 va récupérer un évènement dans sa file, le traiter, ajouter un nouvel événement la file de S3 puis continuer son exécution. Le scénario sera le même jusqu'à l'étage S5.

– (b) *représente une exécution totalement synchrone*. Cette exécution est similaire à celle que l'on peut trouver dans les systèmes multi-threadés. L'étage S1 va appeler (dans son propre thread) le traitant de l'étage S2. Il s'agit là d'un appel de fonction ordinaire. Le traitant de l'étage S2 va appeler le traitant de l'étage S3, toujours avec le thread de l'étage S1.
– (c) *représente une exécution partiellement synchrone*. Les appels de S1 à S2 et de S4 à S5 sont effectués de manière synchrone, alors que les appels de S2 à S3 et de S3 à S4 sont effectués de manière asynchrone. Cela est similaire à ce qui est fait dans Click lorsque le programmeur introduit un point de découplage.

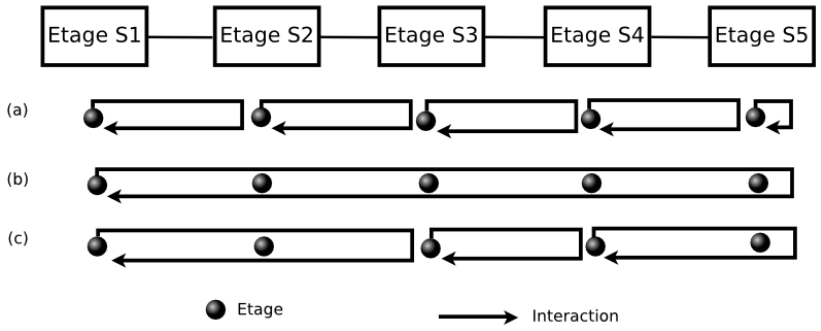

FIG. 2.2 – Modèles d'exécution asynchrone, synchrone et partiellement synchrone

2.2.2 Architecture

Notre objectif à terme étant d'intégrer notre support d'exécution au modèle de composants Fractal, nous utilisons la terminologie des modèles de composants pour décrire l'architecture du système proposé. Chaque étage est un composant ayant au moins deux interfaces en entrée, une pour les appels synchrones et une pour les appels asynchrones, ainsi que deux interfaces en sortie. Les interfaces de sortie synchrones sont reliées à des interfaces d'entrées synchrones et inversement. Si l'étage dispose de plusieurs files, par exemple pour séparer différents flux, alors le composant disposera de plusieurs interfaces d'entrée asynchrones. Si un étage choisit de faire un appel synchrone, ce dernier va appeler directement le traitant d'évènement de l'étage suivant via le point d'entrée synchrone. À l'inverse si ce même étage décide d'appeler l'étage suivant de manière asynchrone, son message sera mis dans la file des messages en attente de l'étage et sera traité ultérieurement.

Un étage est un composant composite qui est constitué de sous-composants (voir figure 2.3). Ceux-ci sont au nombre de quatre. Un premier composant est utilisé pour la gestion de la file de messages. Ce composant est connecté à un autre composant gérant un groupe de threads. Enfin, deux autres composants sont utilisés : l'un pour le traitement de l'événement et l'autre pour la prise de décision quant au mode utilisé pour la suite de l'exécution. Dans la suite de cette section, nous décrivons ces composants en détail.

Le composant File est utilisé pour gérer la file d'entrée d'un étage. Lorsque les messages

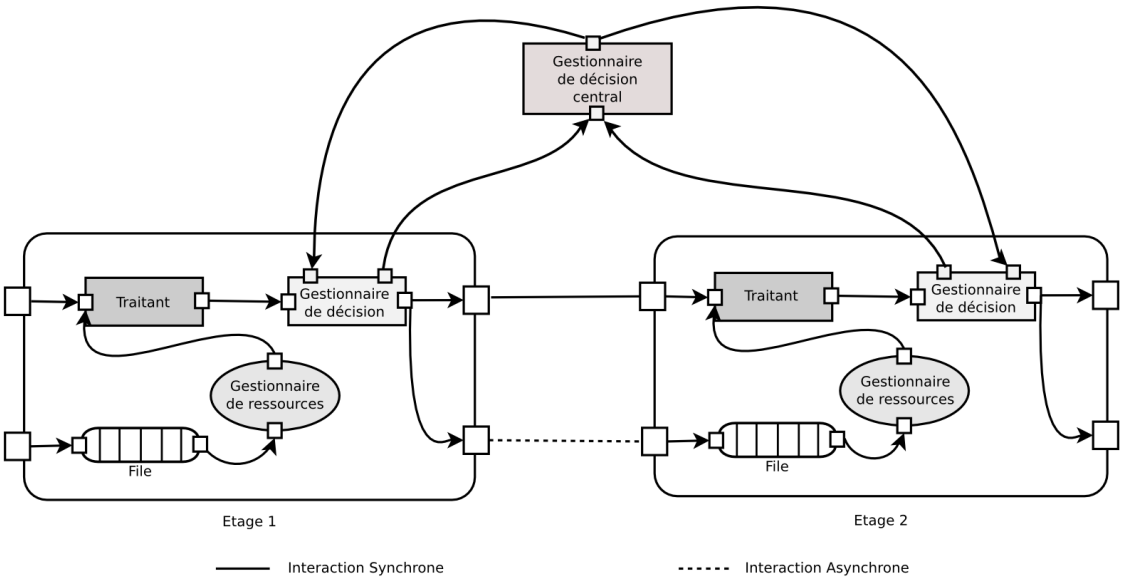

FIG. 2.3 – Architecture des étages

arrivent par l'entrée asynchrone, ils sont ajoutés à cette file dans le but d'être traités ultérieurement. Différents types de files peuvent être utilisées selon le comportement voulu. Il est par exemple possible de définir un file de taille non bornée qui grossit dynamiquement. Il est également possible d'avoir une file de taille bornée. Dans ce cas là, lorsqu'un étage veut mettre un message dans la file et que cette dernière est pleine, soit l'ajout est bloquant jusqu'à ce qu'il y ait de la place dans la file soit, si la sémantique de l'application le permet, le message est perdu. Enfin, il peut être possible de combiner ces caractéristiques avec une politique de contrôle d'admission selon le comportement voulu.

Chaque étage dispose de ses propres ressources d'exécution. Cela est modélisé par un composant gestionnaire de ressources. Ce composant permet de gérer un ensemble de threads traitant les message présents dans la file de l'étage. La taille de ce groupe de threads peut évoluer en fonction de la charge courante. Toutefois, cette évolution doit se faire de manière progressive pour éviter qu'une charge fluctuante entraîne de trop gros coûts de création/destruction de ressources. Une autre amélioration est le principe de traitement par lot. Chaque thread récupère, si possible, plusieurs événements à traiter. Cette technique est déjà présente dans SEDA et StagedServer : elle permet d'améliorer l'utilisation des caches présents sur la machine. Un des inconvénients de cette technique est que si le système est très peu chargé (le temps entre les messages est supérieur au délai de garde de l'opération de récupération de messages) le temps de traitement d'un message augmente. Il doit donc être possible de paramétrer les opérations de traitement par lots. Pour récupérer les événements, deux politiques sont possibles. Soit le thread s'endort et est réveillé par la file lorsqu'un nouveau (groupe de) message(s) a été ajouté dans la file, soit le thread vérifie périodiquement l'arrivée de messages dans la file. La première solution a été choisie car elle semble à priori moins coûteuse en ressources. En effet dans le second cas, à faible charge, il est possible que certains threads soit réveillés alors qu'il n'y a aucun événement dans la file et consomment donc des cycles processeurs (plus des coûts de changements de contexte) pour rien.

Le troisième composant est le traitant d'événements. Ce traitant effectue certaines opérations en fonction de l'évènement à traiter. Il peut être appelé de deux manières : soit par un thread du groupe de threads, soit directement par l'entrée synchrone. Le fait que plusieurs threads puissent appeler simultanément le traitant d'événement oblige le programmeur à s'assurer que les accès concurrents à la mémoire ne seront pas problématiques. Toutefois, ce problème étant confiné au niveau du traitant, il est généralement assez aisé de le résoudre.

Enfin, le dernier composant constituant un étage est le gestionnaire de décision. Il s'agit du dernier composant de la chaîne ; son rôle est de décider si l'appel au prochain étage doit s'effectuer de manière synchrone ou de manière asynchrone. La solution qui a été choisie est de mettre un gestionnaire de décision au niveau de l'étage lui même. Ces gestionnaires de décision sont coordonnés par un gestionnaire de décision global. Cela permet de garantir une cohérence dans les décisions prises et notamment dans le cas d'un passage dans un modèle d'exécution totalement synchrone/asynchrone. Outre le gestionnaire de décision global, deux acteurs interviennent dans la prise de décision : l'étage qui possède le thread et l'étage courant. Il est nécessaire de faire intervenir l'étage possédant le thread. En effet, dans le cas contraire il est possible que le thread de cet étage soit monopolisé par les étages suivants sans possibilité pour l'étage possédant le thread de le récupérer. Cette interaction est illustrée sur la figure 2.4.

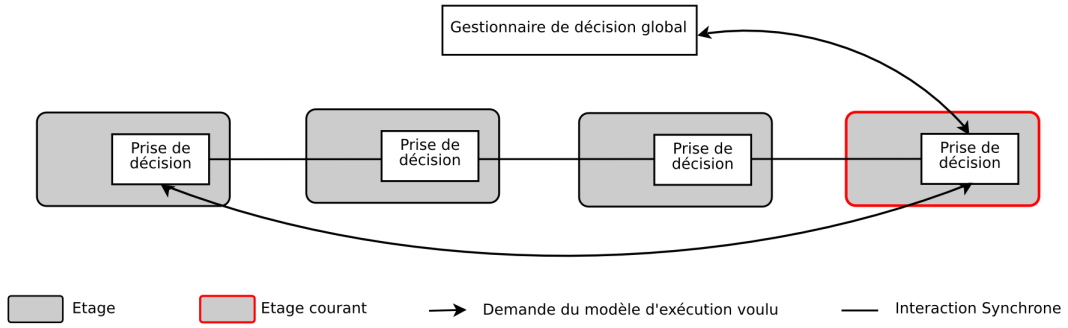

FIG. 2.4 – Interaction des étages pour la prise de décision

Pour que la prise de décision soit possible, chaque message est constitué de deux parties : un pointeur vers l'étage possédant le thread courant et le message en lui même. Cette référence est passée d'étage en étage dans le cas d'une exécution synchrone. Lors d'une exécution asynchrone la référence pointe sur l'étage lui même et dans ce cas, il est le seul à prendre la décision d'effectuer la suite en synchrone ou en asynchrone. Le gestionnaire de décision va donc interroger l'étage courant et l'étage qui possède le thread pour savoir si l'appel au prochain étage se fera de manière synchrone ou de manière asynchrone. Si les deux étages veulent continuer en synchrone, l'appel sera synchrone. Si un des deux veut qu'il soit asynchrone, c'est cette décision qui sera prise. La priorité est donc donnée à l'asynchrone. Ceci est essentiellement fait pour assurer que l'étage possesseur du thread puisse le récupérer dès que cela s'avère nécessaire.

2.2.3 Prise de décision

Dans la partie précédente, nous avons parlé de l'architecture proposée, ainsi que de la manière dont le gestionnaire de décision est intégré au système. Cependant, les différentes heuristiques nécessaires pour la prise de décision n'ont pas encore été abordées. Les éléments qui semblent importants pour faire un choix sont :

- La durée moyenne de l'exécution d'un étage
- La taille de la file d'entrée d'un étage
- Le nombre de fautes de cache
- Un nombre important de changements de contexte
- La nature bloquante des opérations effectuées par un étage

La durée moyenne d'un étage est un paramètre important. Si la durée de l'étage est trop courte, le ratio *durée de l'étage/ temps d'un changement de contexte* sera mauvais. Il sera donc intéressant d'agréger l'exécution de plusieurs étages pour améliorer ce ratio. C'est par exemple le cas dans Click où les éléments sont très courts et leur interaction synchrone.

La taille de la file d'un étage donne également des indications pour prendre la décision. Si la taille de cette file augmente, c'est qu'il n'y a pas assez de ressources disponibles pour traiter les événements entrant dans cet étage. Deux solutions existent. La première consiste à allouer de nouvelles ressources d'exécution pour l'étage. La seconde est de dire par l'intermédiaire du gestionnaire de décision que le mode d'exécution des threads de cet étage doit devenir synchrone. A travers cet exemple, la nécessité d'une coordination entre le gestionnaire de ressources et le gestionnaire de décision est clairement mise en avant.

L'un des avantages de la programmation à base d'étages est de pouvoir améliorer l'utilisation des caches de la machine grâce au traitement par lot qui permet de regrouper les opérations manipulant les mêmes données. Passer à un modèle d'exécution synchrone enlève dans la plupart des cas cet avantage. Si le nombre de défauts de cache est important, il peut être intéressant de repasser l'exécution dans un modèle asynchrone. Toutefois, dans le cas où les données en cache ne sont pas dépendantes du traitant mais de la requête, une exécution synchrone sera plus intéressante qu'une exécution asynchrone. Un exemple peut être un service appliquant différents filtres sur une image, chaque étage représentant un filtre. Dans ce cas là, les données qui sont manipulées au niveau de chaque étage correspondent au message qui est passé entre les étages (l'image). Pour améliorer les effets de caches il faut donc que tout le chemin traitant cette image soit fait en synchrone. En effet, en asynchrone l'exécution manipulerait des images différentes à chaque fois et donc ne pourrait pas maximiser l'utilisation des données déjà en cache.

Le nombre de changements de contexte est un paramètre intéressant à regarder puisqu'une exécution asynchrone génère plus de changements de contexte qu'une exécution synchrone. Comme expliqué précédemment, si le nombre de changements de contexte est trop important, le coût de ces derniers sera conséquent.

Enfin, un autre paramètre pour effectuer le choix peut être le fait qu'un état réalise des opérations bloquantes comme une E/S synchrone. Dans ce cas, comme le fait Flash, il est intéressant d'appeler cet étage de manière asynchrone, ce qui revient à fournir des E/S asynchrones même si le système sous-jacent ne le supporte pas.

2.3 Synthèse

Dans ce chapitre, nous avons présenté notre contribution qui consiste à améliorer les modèles de programmation par étages afin de rendre leur modèle d'exécution dynamiquement adaptable. Nous avons décrit une architecture basée sur une modification de Sandstorm, une implantation Java de SEDA. Par ailleurs, nous avons présenté différents paramètres qui peuvent être utilisés pour choisir dynamiquement le modèle d'exécution le mieux approprié.

Chapitre 3

Mise en oeuvre

Nous avons décidé de baser nos travaux sur l'implantation en Java de SEDA : Sandstorm. Ce choix a été fait essentiellement pour des raisons de temps. SEDA est un modèle architectural pour la conception de services internet robustes qui a été développé à l'Université de Berkeley. SEDA définit une architecture basée sur des étages pour permettre de réaliser des serveurs tolérant des charges importantes. Nous expliquons tout d'abord la manière dont le code de Sandstorm est structuré. Puis dans une seconde partie, nous décrivons les modifications que nous avons effectuées sur Sandstorm. Dans cette étude, nous nous intéressons essentiellement au domaine applicatif des serveurs de données. Ce choix a été fait car ce domaine a fait l'objet de nombreuses recherches et différents comportements ont été décrits.

3.1 Architecture de Sandstorm

Comme expliqué précédemment SEDA est une architecture pour faire de la programmation à base d'étages. Divers projets, tels que l'ESB (Entreprise Service Bus) libre Mule[1], se basent sur Sandstorm et ses différentes bibliothèques. Sandstorm est entièrement écrit en Java mais n'est plus maintenu depuis Juillet 2002. Il est donc compatible avec les différentes versions de Java depuis Java 1.3 mais ne permet pas de disposer des différentes améliorations apportées depuis. L'une de ces amélioration concerne les E/S grâce au paquetage NIO[2], présent depuis la JDK[3] 1.4, qui permet notamment de disposer d'appels non bloquants. Un autre amélioration présente dans Java depuis concerne la gestion de la concurrence, grâce au paquetage *java.util.concurrent* qui permet de disposer de nouvelles primitives de synchronisation et de diverses améliorations notamment dans la gestion des groupes de threads. La plupart de ces fonctionnalités sont cependant fournies directement par Sandstorm, notamment des classes *ThreadPool* (pour la gestion des groupes de threads) ou le paquetage NBIO pour la gestion des E/S non bloquantes.

Étages

Les étages de Sandstorm sont modélisés par deux classes. La première est la classe *StageIF* qui représente l'étage vu par les autres étages (point de vue externe). *StageIF* permet d'accéder

[1] http ://mule.codehaus.org/display/MULE/Home
[2] new I/O
[3] Java development Kit, kit de développement Java

aux files de l'étage. Par défaut les étages n'ont qu'une seule file visible de l'extérieur. La classe *StageWrapperIF* représente, quant à elle, l'étage vu ses composants (point de vue interne), c'est à dire que ceux-ci peuvent accéder à la file de l'étage mais aussi au gestionnaire de threads et au traitant d'événements.

D'un point de vue extérieur à l'étage, la file est modélisée par l'interface *QueueIF* qui spécifie les différentes méthodes pour ajouter des éléments dans une file (ajout bloquant ou non, ajout de plusieurs éléments, etc.). Par ailleurs, une interface *SourceIF* permet de manipuler la file d'un point de vue interne à l'étage, c'est à dire faire les opérations de récupération de messages.

Le gestionnaire de threads est représenté par une classe *ThreadManagerIF* qui permet de gérer automatiquement les groupes de threads (ajout, retrait, etc.). Chaque thread du groupe effectue une boucle dans laquelle il récupère des événements à traiter. Il s'agit là d'une différence par rapport à notre architecture dans laquelle les threads sont réveillés par la file lorsqu'il y a des éléments dans cette dernière.

Enfin, notons que les étages sont configurés et initialisés au moyen d'un fichier de configuration. Il est possible notamment de leur passer des paramètres à l'initialisation ou encore de définir le nombre de threads dédiés à cet étage.

Entrées / Sorties

Sandstorm utilise son propre paquetage NIO, appelé *NBIO*[4] pour pouvoir disposer d'E/S non bloquantes[5]. La gestion des sockets est effectuée par la classe statique *ASocketMgr*. Cette classe instancie trois étages représentés par les classes *ListenStage*, *ReadStage* et *WriteStage*. Disposer d'une classe statique permet de garantir qu'il n'y aura qu'une seule instance de ces étages pour toute l'application. Cette classe dispose d'une méthode *EnqueueRequest* qui va, selon la nature de l'application, mettre le message dans la file de l'étage concerné. Par exemple, s'il s'agit d'une requête d'écriture, le message sera mis dans la file de l'étage d'écriture (*WriteStage*).

Chacun de ces étages dispose de deux files : une file qui permet d'interagir avec NIO et une file qui est utilisée pour la communication entre étages. Du fait qu'il y ait ces deux files de messages, le gestionnaire de threads utilisé ici est différent de celui utilisé pour les autres étages. Chaque thread du gestionnaire va d'abord essayer de récupérer des messages de la part de NIO. Cette récupération peut être bloquante ou non. Si elle n'est pas bloquante il est possible que l'exécution boucle tant qu'elle n'a rien trouvé. Après avoir appelé le traitant sur ces messages, le thread va ensuite récupérer les messages en provenance d'autres étages et appeler les traitants associés. A la fin, un changement de contexte est forcé pour éviter que le thread boucle trop rapidement s'il n'y a pas de messages à récupérer. Cette solution est coûteuse, ce qui justifie notre choix d'une architecture dans laquelle les threads sont réveillés par les files.

La figure 3.1 illustre un serveur utilisant ces étages. L'étage *SimpleTCPServer* a pour rôle d'initialiser les sockets, et de faire passer les messages entre les étages d'E/S. Les connections TCP sont représentées par des objets *ATcpConnection*. La sémantique de ces objets est un peu étonnante. En effet, un objet ATcpConnection est à la fois une file (il est possible d'appeler des

[4]Non Blocking I/O

[5]En effet, Sandstorm étant basé à l'origine sur Java 2 version 1.3, cette dernière ne proposait pas de méthode permettant d'utiliser de telles E/S. Cependant depuis Java 2 version 1.4, le paquetage NIO est proposé par la JDK. Sandstorm propose donc aussi une méthode pour utiliser NIO à la place de NBIO.

opérations telles que *enqueue*) et un événement à passer entre les étages. La méthode *enqueue* de cet objet fait elle-même appel aux méthodes *enqueue* de ASocketMgr. L'empilement des appels, et la possibilité pour un étage d'utiliser cette méthode *enqueue* à tout moment, rend difficile la compréhension générale du code et s'éloigne du modèle architectural proposé où les étages sont indépendants et les relations entre les étages explicites.

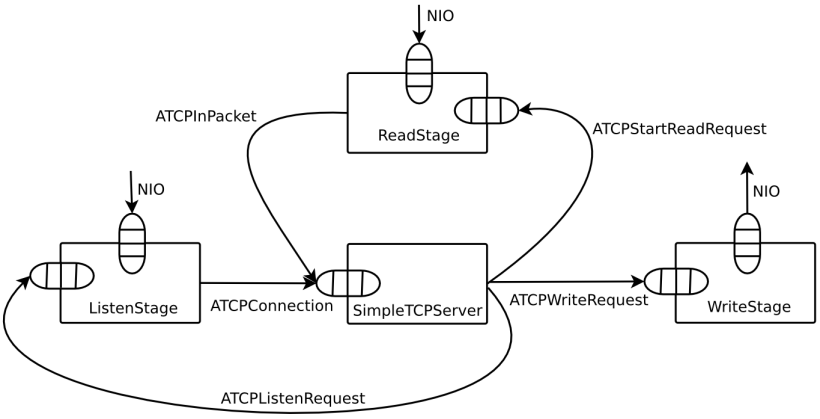

FIG. 3.1 – Un exemple d'utilisation des étages de sockets

Notons qu'une limitation imposée aux étages de sockets est de n'avoir qu'un seul thread par étage (cf section 3.2). Si ce choix n'est pas très problématique dans un cas d'exécution asynchrone, il peut vite s'avérer gênant lorsque l'exécution est totalement synchrone. En effet, si le premier étage ne dispose que d'un seul thread, dans le cas d'une exécution totalement synchrone cela signifie que l'application n'utilisera que ce seul et unique thread.

Enfin, nous avons jusqu'ici essentiellement parlé des E/S réseau. Les E/S disques sont implémentées selon le même principe et souffrent des mêmes défauts. Par ailleurs, les E/S défaut ont un défaut supplémentaire : il n'y a pas d'étage. En effet, tout les traitements sont effectués par le gestionnaire de threads (récupération des opérations à effectuer, prise en charge effective des opérations en utilisant des E/S synchrones, envoi de messages aux étages concernés, etc.).

Haboob & le protocole HTTP

Sandstorm propose une bibliothèque permettant d'utiliser le protocole HTTP. La couche HTTP est composée d'un étage et de différents messages (nouvelle connexion, nouvelle requête, etc.). Cet étage a pour rôle de récupérer les messages venant de la couche TCP, d'en extraire les informations (entêtes, corps de la requête, etc.) et de les transmettre à l'étage suivant. Comme pour les connexions TCP, les connexions HTTP sont modélisées par un objet *HttpConnection*. Une fois de plus, cet objet est à la fois une file et un événement. On augmente ainsi encore le nombre d'appels effectués qui ne sont pas maîtrisés par le programmeur de l'application car réalisés par des étages "masqués". Notons, par ailleurs, que l'étage HTTP ne peut pas être initialisé comme les autres étages de l'application : il n'est par exemple pas possible de spécifier le nombre de threads dédiés à cet étage.

La couche HTTP a été utilisée pour construire Haboob, un serveur Web utilisé pour prouver l'efficacité de l'architecture SEDA. Haboob utilise la bibliothèque HTTP et utilise les

E/S disque asynchrones. La figure 3.2 illustre l'architecture de Haboob. Haboob est composé de 3 étages, plus les étages liés à HTTP et ceux liés aux E/S disques.

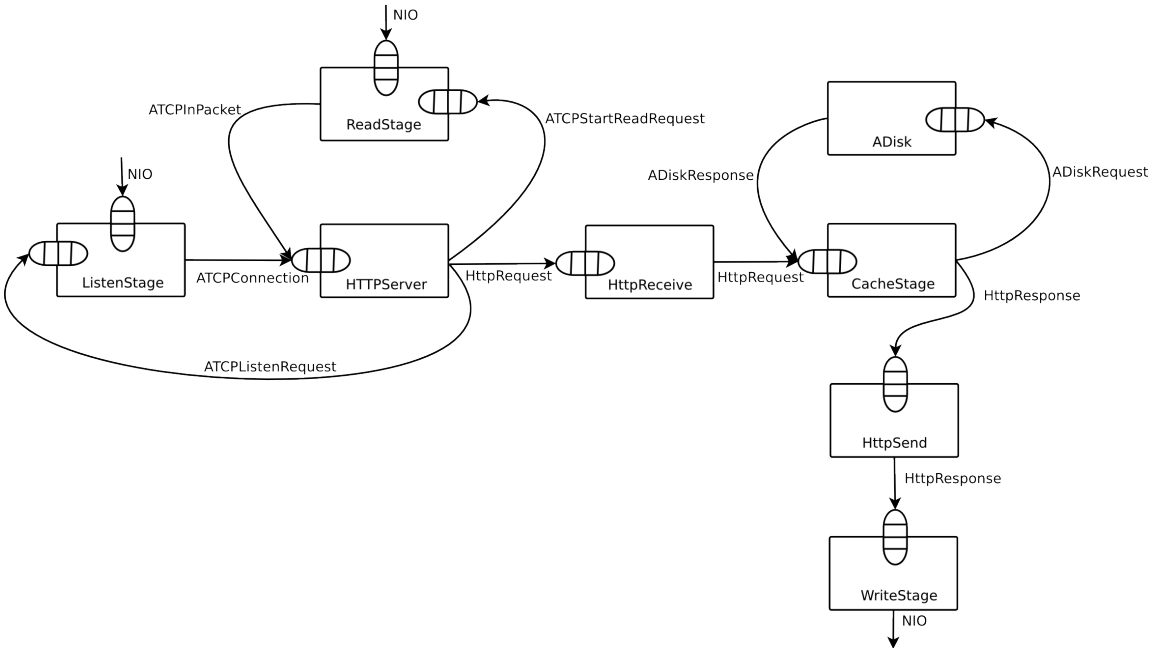

FIG. 3.2 – L'architecture de Haboob

3.2 Modifications apportées à Sandstorm

Dans cette partie nous présentons les modifications que nous avons apportées à Sandstorm. Notre objectif était de rendre le modèle d'exécution dynamiquement adaptable, et ce de façon transparente vis-à-vis du code des applications.

Caller & Abstact Handler

La première modification que nous avons appliquée à Sandstorm a consisté à rajouter une interface *Caller* (voir figure 3.3). Cette interface permet d'étendre les traitants en leur ajoutant deux opérations pour :

- Demander au gestionnaire de décision s'il veut que la suite de l'exécution soit synchrone,
- Envoyer un message à l'étage suivant. C'est cette fonction qui interroge les différents intervenants pour décider si l'appel doit être synchrone ou asynchrone

Nous avons également introduit la classe abstraite *AbstractHandler* qui est un traitant d'événements abstrait implémentant l'interface *Caller* et la méthode *send*. La figure 3.4 présente une partie du code de la classe *AbstractHandler*. Nous pouvons notamment voir que lors de la méthode *send*, l'étage courant et l'étage possédant le thread sont interrogés. Ensuite, soit le message est mis dans la file de l'étage suivant, soit le traitant de cet étage est directement appelé.

Il a donc été nécessaire d'ajouter une méthode de décision au niveau de l'interface *StageWrapper* qui est la représentation interne d'un étage. Un mot clé `sync` a également été rajouté.

```
package seda.sandStorm.api;

public interface Caller
{
  public void send(Caller caller, QueueElementIF item, StageIF stage)
     throws EventHandlerException, SinkException;
  public boolean synchronousInteraction();
  public String getCallerName();
}
```

FIG. 3.3 – L'interface Caller

Ce mot clé permet de définir manuellement via le fichier de configuration si l'étage veut une exécution initialement synchrone ou asynchrone. Le paramètre *caller* a été rajouté pour la méthode *HandleEvent*. Ce paramètre est nécessaire pour connaître quel est l'étage possesseur du thread, afin de l'interroger par la suite. Comme nous avons rajouté un paramètre, il a donc été nécessaire de modifier les appels à cette fonction effectués par le gestionnaire de threads. Lorsque c'est le gestionnaire de threads qui fait l'appel à la méthode du traitant, l'exécution est de type asynchrone (cf figure 2.3). Dans ce cas, le possesseur du thread (l'appelant) n'est autre que l'étage lui même.

Entrées / Sorties

Pour effectuer les E/S disque, Sandstorm utilise la paquetage NIO présent depuis Java 2 version 1.4. Comme il a été dit, Sandstorm impose une limite d'un seul thread pour les étages manipulant les sockets. Il a donc été premièrement nécessaire d'éliminer cette restriction pour pouvoir avoir une exécution totalement synchrone. Le premier travail a été de déterminer quelles étaient les sections dans le code des sockets qui devaient être mises en exclusion mutuelle, pour éviter, par exemple, d'avoir plusieurs écritures concurrentes simultanées. Un des problèmes qui est apparu au niveau de la gestion des sockets est que le code est très lourd et complexe à maîtriser. Les répercussions des verrous sur les performances sont difficilement analysables.

Afin d'évaluer le gain apporté par les modifications effectuées dans le but de permettre l'ajout de threads pour les étages manipulant les sockets, nous avons réalisé l'expérience suivante : un client envoie une requête d'un octet au serveur TCP qui la lui renvoie. Nous disposons de 7 noeuds clients. Selon les cas, chaque noeud héberge entre 1 et 20 processus clients. La figure 3.5 montre les gains de performance amenés par l'utilisation de plusieurs threads. On remarque que le gain apporté par l'utilisation de plusieurs thread n'est pas très convaincant. Nous pensons que ceci est dû à la complexité du code des étages de sockets, ainsi qu'au coût des opérations de synchronisation.

Nous avons donc décidé de fournir nos propres étages de sockets. Contrairement à l'implantation de base, nous n'utilisons que deux étages : un étage de lecture et un étage d'écriture. Cette nouvelle architecture est représentée sur la figure 3.6. Les threads du gestionnaire de threads de l'étage de lecture se bloquent tous en attente de l'acceptation d'une connexion. Lorsqu'une connexion arrive, le thread qui l'accepte est responsable de la gestion de cette connexion jusqu'à ce qu'elle soit fermée. Ensuite, le thread revient se bloquer en attente d'une nouvelle connexion. L'étage de lecture est donc un étage particulier dans le sens où il

```
package seda.sandStorm.core;

import seda.sandStorm.api.Caller;

public abstract class AbstractHandler implements EventHandlerIF, Caller
{
  protected StageWrapperIF wrapper;
  protected String name;
  protected boolean sync;

  public final void init(ConfigDataIF config) throws Exception
  {
    //Initialisation
    wrapper = config.getStage().getWrapper();
    name = wrapper.getStage().getName();
    sync = (config.getString("sync") != null);
    wrapper.setSynchronousMode(sync);

    doInit(config);
  }

  [...]

  public void send(Caller caller, QueueElementIF item, StageIF stage)
    throws EventHandlerException, SinkException{
      if (caller.synchronousInteraction()
          && this.wrapper.synchronousInteraction()){

        // La decision est d'effectuer l'appel en synchrone
        // Appel direct au traitant du prochain etage
        stage.getWrapper().getEventHandler().handleEvent(caller, item);
      }
      else{
        // La decision est d'effectuer l'appel en asynchrone
        // Ajout du message dans la file du prochain etage
        stage.getSink().enqueue(item);
      }
  }

  public boolean synchronousInteraction(){
    return wrapper.synchronousInteraction();
  }
}
```

FIG. 3.4 – La classe abstraite AbstractHandler

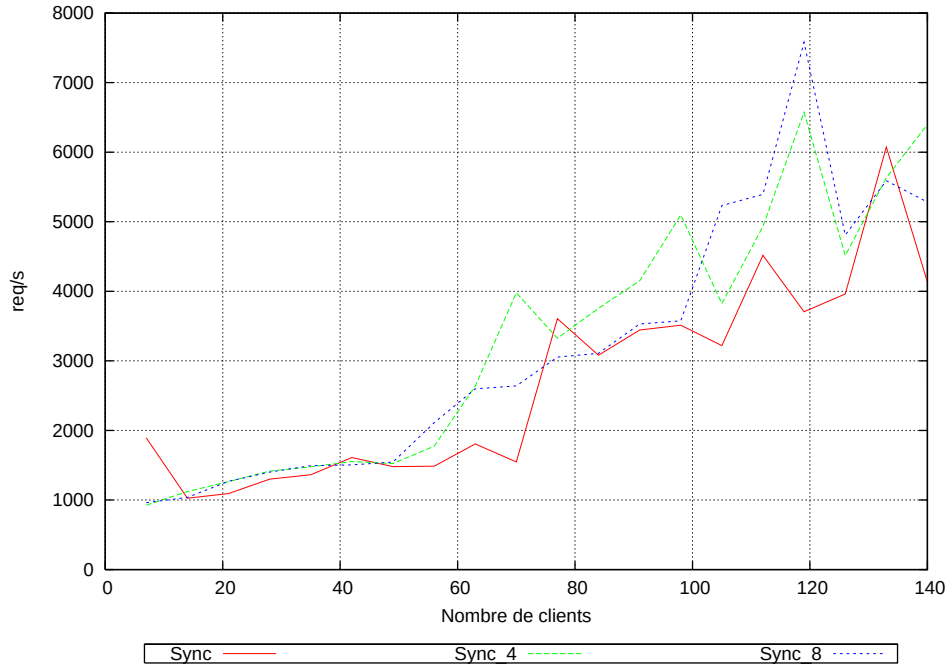

FIG. 3.5 – Comparaison des débits obtenus en augmentant le nombre de threads sur un serveur utilisant la version modifiée des étages de sockets de Sandstorm

n'a pas de file de messages en entrée mais est interfacé directement avec les couches réseau du système. Notons, enfin, qu'avec cette nouvelle architecture, il est possible de spécifier le nombre de threads initial de chaque étage.

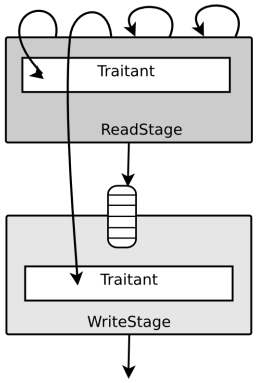

FIG. 3.6 – Nouvelle architecture de sockets

Les threads

Dans le but de permettre un changement d'une exécution entièrement asynchrone vers une exécution entièrement synchrone, nous avons modifié la gestion des threads à l'intérieur de Sandstorm. En effet, comme le montre la figure 3.7, il est nécessaire de rééquilibrer les threads présents dans les différents étages. Pour ce faire, nous avons dû modifier la gestion de threads pour permettre de changer l'association thread/étage et pour pouvoir affecter dynamiquement un thread à un autre étage. L'objectif de cette ré-allocation dynamique est d'éviter des coûts importants de création/destruction de threads. Divers problèmes ont été rencontrés : le premier concerne le fait de devoir vider les files de messages avant de passer à un mode d'exécution entièrement synchrone. Le second problème concerne les appels bloquants (tel que l'appel *accept* en Java). En effet, lorsque le thread est bloqué sur une instruction, il n'est pas possible de changer son affectation. Nous avons résolu ce problème en réveillant périodiquement le thread pour qu'il vérifie s'il ne doit pas changer d'étage. Une autre solution à étudier est d'utiliser des signaux pour prévenir le thread bloqué.

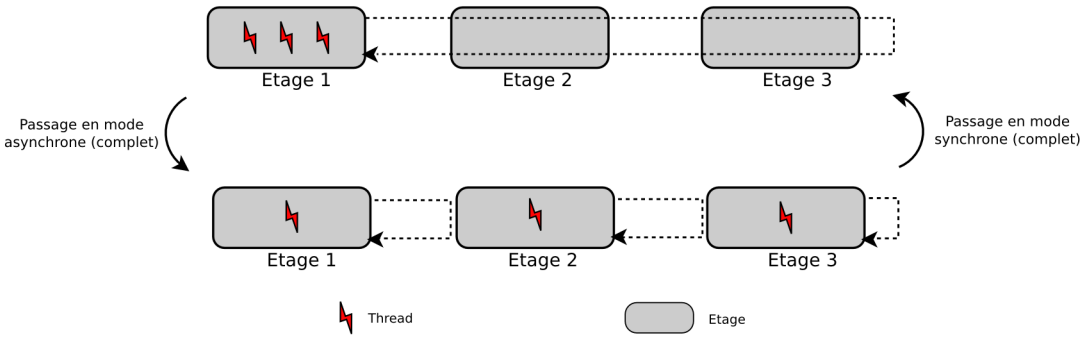

FIG. 3.7 – Rééquilibrage des threads lors d'un changement de modèle d'exécution

3.3 Synthèse

Dans ce chapitre, nous avons décrit l'implantation de l'architecture proposée au chapitre 2. Cette implantation repose sur Sandstorm, l'implantation Java de SEDA. Sandstorm a été modifié afin de rendre le modèle d'exécution dynamiquement adaptable, et ce de façon transparente vis-à-vis du code des applications.

Chapitre 4

Résultats expérimentaux

Dans ce chapitre, nous présentons les résultats obtenus avec la nouvelle architecture décrite dans le chapitre précédent. Dans ces tests, nous nous intéressons à la comparaison de deux modèles d'exécution : l'un totalement synchrone et l'autre totalement asynchrone. L'objectif de ces tests est de montrer qu'il existe des cas où l'utilisation de l'un ou l'autre de ces deux modèles entraîne des différences de performance, et donc de vérifier que le principe de changement dynamique de modèle d'exécution a un sens. Les premiers tests ont été effectués sur le serveur web fourni par Sandstorm : Haboob. Les résultats de ces tests ne sont pas présentés dans ce rapport. En effet, nous avons eu de nombreux problèmes de stabilité que nous ne sommes pas parvenus à résoudre rapidement. Nous avons donc effectué des tests sur un serveur simplifié. Dans une première partie, nous présentons les conditions expérimentales utilisées. Nous présentons ensuite les tests effectués et les résultats obtenus. Les premiers tests présentés ne font intervenir que les deux étages de sockets. Les tests suivants utilisent un troisième étage simulant un goulot d'étranglement. Enfin, le troisième test illustre la capacité de modification dynamique du modèle d'exécution.

4.1 Conditions expérimentales

4.1.1 Injection de charge

Nous avons crée un injecteur de charge en boucle fermée. Les injecteurs de charge en boucle fermée effectuent en boucle le schéma suivant : (1) Envoi d'une requête, (2) Réception de la réponse, (3) Attente d'un quantum de temps. L'injecteur de charge que nous avons développé utilise un thread par client. Initialement, cet injecteur était écrit en Java. Toutefois, nous avons remarqué d'importants problèmes de performances. Nous avons donc réécrit cet injecteur en C. Ce dernier donne des performances plus proches de celles attendues. L'injecteur de charge fonctionne en émettant une requête par connexion. Chaque client ouvre une connexion vers le serveur, envoie une requête, attend la réponse et ferme la connexion. Ce schéma est répété N fois dans le but d'obtenir des moyennes fiables (N varie selon le nombre de client et se situe entre 2000 et 30000 itérations).

Notons que ce type d'injecteur permet de simuler un client qui récupère une page d'un site internet, la parcourt, clique sur un lien et récupère une nouvelle page. Notons également que comme l'expliquent Pariag *et al.* [20], un injecteur de charge en boucle fermée ne permet pas toujours de mettre le système dans un état de surcharge, surtout si le nombre de nœuds clients est faible. Une autre manière d'injecter de la charge est d'envoyer les requêtes sans

se soucier des réponses, en maintenant un certain taux de requêtes par seconde. Ce modèle est l'injection de charge à boucle ouverte. L'injecteur est alors composé de deux threads, l'un s'occupant de l'envoi de message, l'autre de la réception. Pour des raisons de temps, nous n'avons pas utilisé de tel injecteur dans les travaux présentés dans ce rapport.

4.1.2 Configuration de test

Nous avons effectué les tests sur des machines ayant les caractéristiques suivantes :
- Deux processeurs Intel Xeon avec la technologie Hyperthreading
- Une carte réseau ethernet d'un débit de 1Gb/s
- Un disque dur IDE
- Un noyau Linux 2.6.20

Ces machines sont reliées par un commutateur réseau non bloquant. Nous avons vérifié au moyen de l'outil `netperf` que le débit réel était proche du débit théorique. Nous disposons au total de 7 machines pour effectuer nos tests. Une machine sur les 7 étant utilisée comme serveur, nous disposerons donc de 6 noeuds clients pour injecter de la charge. Pour pouvoir comprendre et analyser les résultats obtenus, nous avons choisi de désactiver l'option *SMP*[1] du noyau Linux pour la machine serveur. Ainsi le système ne verra qu'un seul processeur au lieu de 4. L'étude de l'intérêt des architectures parallèles fera l'objet de travaux futurs.

4.2 Tests simples

Dans cette partie, nous présentons les résultats obtenus sur un test simple. Ce test utilise une configuration simple composée de deux étages. Le premier étage reçoit des messages du réseau. Ces messages sont composés d'un entête qui contient la longueur des données et des données proprement dites. Le premier étage fait passer ce message au second étage qui renvoie une réponse de même taille au client. La communication entre le premier et le second étage peut être effectuée soit de manière asynchrone soit de manière synchrone. Notons que le nombre de threads ne varie pas en fonction de la charge. Par ailleurs, pour les tests en asynchrone nous supposons, pour limiter le nombre de cas, que le nombre de threads par étages est égal d'un étage à un autre.

Les résultats du test sont représentés sur la figure 4.1. La figure (a) présente l'évolution du temps de réponse, tandis que la figure (b) présente l'évolution du débit en fonction du nombre de clients. La version synchrone utilise 16 threads, alors que la version asynchrone utilise 4 threads par étage. Ces paramètres ont été choisis de manière à obtenir les meilleurs performances dans les deux cas.

Nous voyons nettement sur ces courbes que jusqu'à une certaine charge, une exécution synchrone est meilleure qu'une exécution asynchrone. Les gains en débits sont relativement importants à basse charge (jusqu'à 75% de gains en débit par exemple pour 6 clients) pour ensuite se réduire (1 à 2 % à charge moyenne). Les gains en temps de réponse sont aussi élevés (jusqu'à 70% de gains dans certains cas). Cela est dû au fait que le temps de traversée des étages est très court, ce qui rend le coût relatif d'un changement de contexte en asynchrone trop important. Ensuite, le débit du serveur en synchrone chute rapidement.

Un point étonnant est l'augmentation du temps de réponse aux alentours de douze clients. Nous n'avons pour l'instant aucune explication à ce phénomène qui est reproductible. Vérifier

[1] Symmetric MultiProcessing, Architecture permettant d'utiliser plusieurs processeurs simultanément

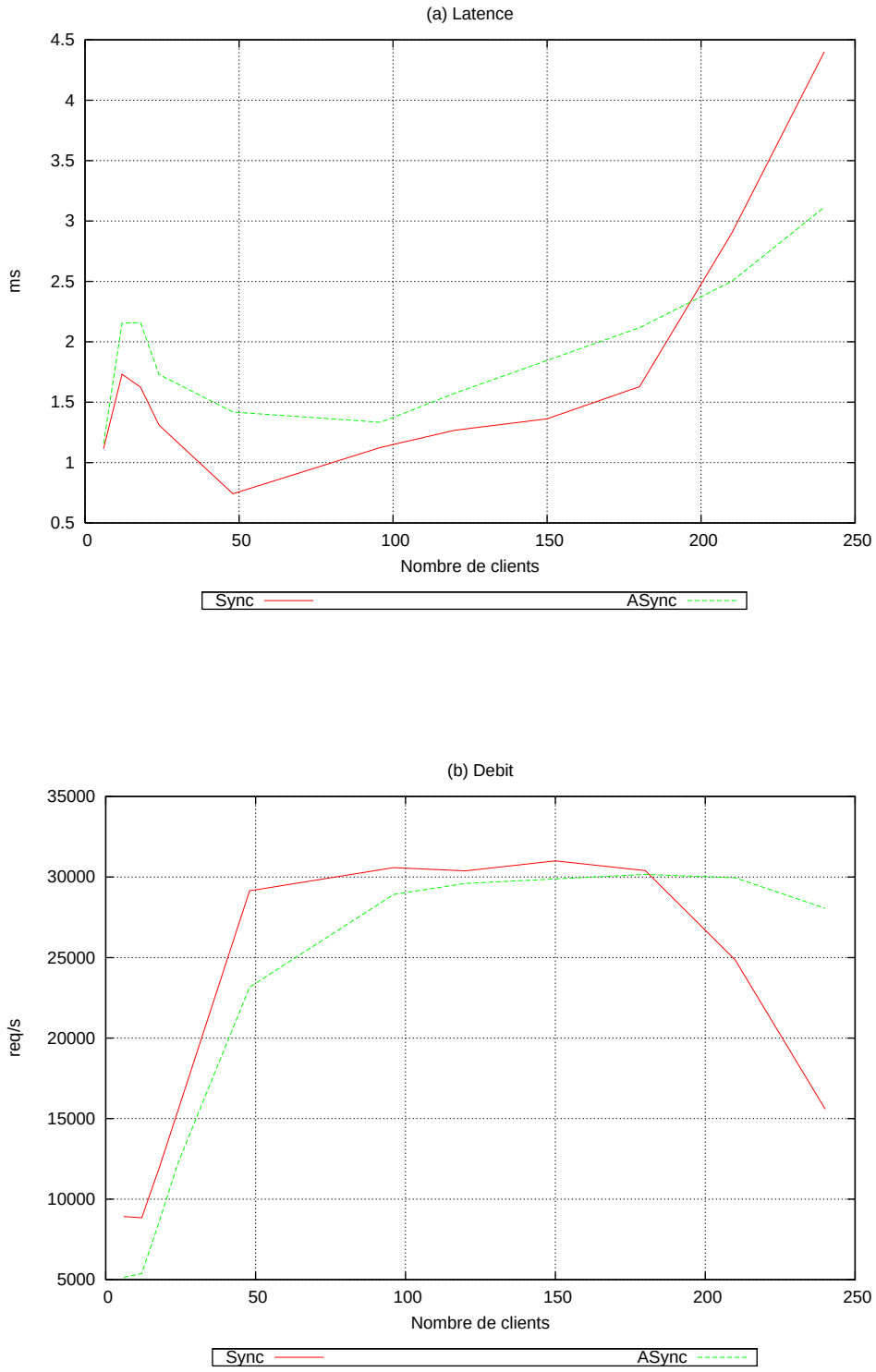

FIG. 4.1 – Comparaison des résultats pour la meilleure exécution en asynchrone et la meilleure exécution en synchrone

si on obtient le même comportement avec une implantation autre que Sandstorm fait partie de nos objectifs futurs.

4.3 Tests avec un goulot d'étranglement

Les résultats obtenus dans la section précédente montrent que dans le cas d'une exécution très courte (i.e. pas d'étages entre les deux étages de socket), une exécution synchrone est nettement plus intéressante qu'une exécution asynchrone à basse charge. Dans cette section, nous présentons des tests dans lesquels nous avons introduit un goulot d'étranglement entre les étages des sockets. Ce goulot d'étranglement est un étage qui effectue dix insertions dans une table de hachage pour chaque événement traité. Cette opération est très coûteuse en termes de temps processeur.

Les résultats obtenus sont présentés sur la figure 4.2. La figure (a) présente l'évolution du temps de réponse, tandis que la figure (b) présente l'évolution du débit en fonction du nombre de clients. La version synchrone utilise 16 threads au total, alors que la version asynchrone utilise 2 threads par étage. Nous pouvons tout d'abord constater que le débit est inférieur aux résultats précédents et que le temps de réponse est lui plus élevé. Pour saturer le serveur, moins de clients sont nécessaires que dans le cas précédent. Nous pouvons également voir sur cette figure que les gains entre la version synchrone et la version asynchrone sont toujours présents (70% pour 6 clients, 28% pour 12 clients, 10% pour 18 clients puis les gains deviennent minimes). Ces gains sont également présents au niveau du temps de réponse (jusqu'à 50%). De façon surprenante, les temps de réponses sont quasiment identiques (1% de différence) à basse charge. Nous estimons que cela est essentiellement dû au fait que le coût introduit par le réseau masque le coût des changements de contexte. Nous pensons donc que sur des types d'applications n'utilisant pas le réseau, les différences entre les versions synchrones et asynchrones pourraient être encore plus marquées.

4.4 Adaptation dynamique du modèle d'exécution

Après avoir montré dans les sections précédentes que le modèle d'exécution synchrone est plus performant que le modèle d'exécution asynchrone à basse charge et vice versa à haute charge, nous présentons dans cette section un mécanisme permettant de changer dynamiquement le modèle d'exécution. L'objectif des tests présentés dans cette section est de montrer que le coût induit par un changement dynamique n'est pas supérieur au gain apporté.

Pour estimer s'il est nécessaire de changer de modèle d'exécution, nous nous sommes intéressés à deux paramètres : le débit d'entrée et le débit de sortie. Le débit de sortie est calculé par l'étage d'écriture. Le débit d'entrée est lui estimé à partir du nombre de débordements du tampon d'acceptation du serveur. Plus précisément, lorsque la couche TCP reçoit une nouvelle demande de connexion, cette demande est passée à l'application dans le but d'être acceptée. Si le serveur ne l'accepte pas immédiatement, elle est mise dans un tampon pour une acceptation future. Le débordement de ce tampon signifie que le serveur n'arrive plus à traiter les demande de connexions entrantes de manière assez rapide et donc que le débit en entrée est élevé. La taille de ce tampon est récupérée en temps réel dans le fichier `/proc/net/netstat`.

L'utilisation conjointe des débits d'entrée et de sortie permet de savoir si le débit de sortie est faible à cause d'un débit d'entrée faible ou à cause d'une saturation du serveur. Dans le premier cas, l'exécution doit s'effectuer de manière synchrone alors que dans le second cas, il

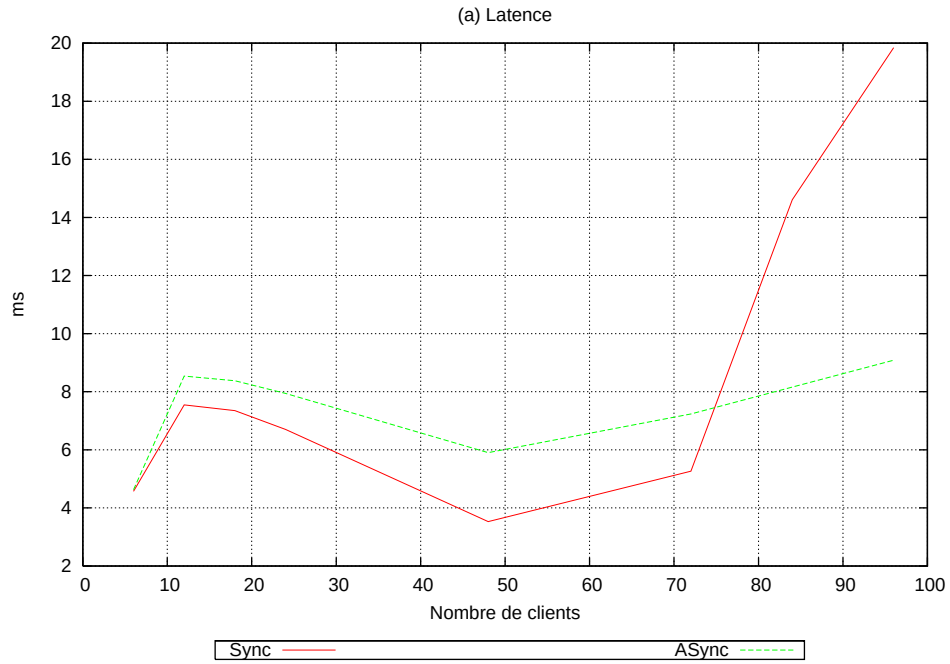

FIG. 4.2 – Comparaison des résultats pour la meilleure exécution en asynchrone et la meilleure exécution en synchrone avec un goulot d'étranglement

faut passer dans un modèle d'exécution asynchrone. Pour éviter des changements de modèles d'exécution inutiles, la décision est prise sur la tendance des 5 derniers échantillonnages. Il y a donc une latence de réaction pour le système. Les valeurs seuils sont actuellement fixées en fonction des observations que nous avons faites dans les sections précédentes. Notre objectif est en effet de montrer la viabilité de la solution, tant au niveau du coût de l'échantillonnage que du mécanisme de rééquilibrage des threads. Dans nos travaux futurs, nous nous intéresserons au réglage dynamique de ces seuils.

La figure 4.3 montre les performances de la version dynamiquement adaptable en comparaison des résultats précédents. Ces tests ont été effectués sur la version du serveur avec goulot d'étranglement. Les résultats obtenus sont très encourageants : nous pouvons voir sur cette figure que le mécanisme d'adaptation permet de changer de modèle d'exécution lorsque cela est nécessaire. Le coût de la décision est relativement faible. En effet, le surcoût ne dépasse pas 5% du débit et 2% du temps de réponse, ce qui est très faible.

4.5 Synthèse

Dans ce chapitre, nous avons présenté des résultats d'expériences réalisées à l'aide de l'architecture que nous proposons. Ces expériences ont tout d'abord permis de confirmer que le modèle d'exécution synchrone était plus performant à basse charge que le modèle d'exécution asynchrone, et vice-versa à haute charge. Ces expériences ont également permis de valider le mécanisme de modification dynamique du modèle d'exécution. Elles ont notamment mis en avant le fait que le surcoût lié à la prise de décision est très faible.

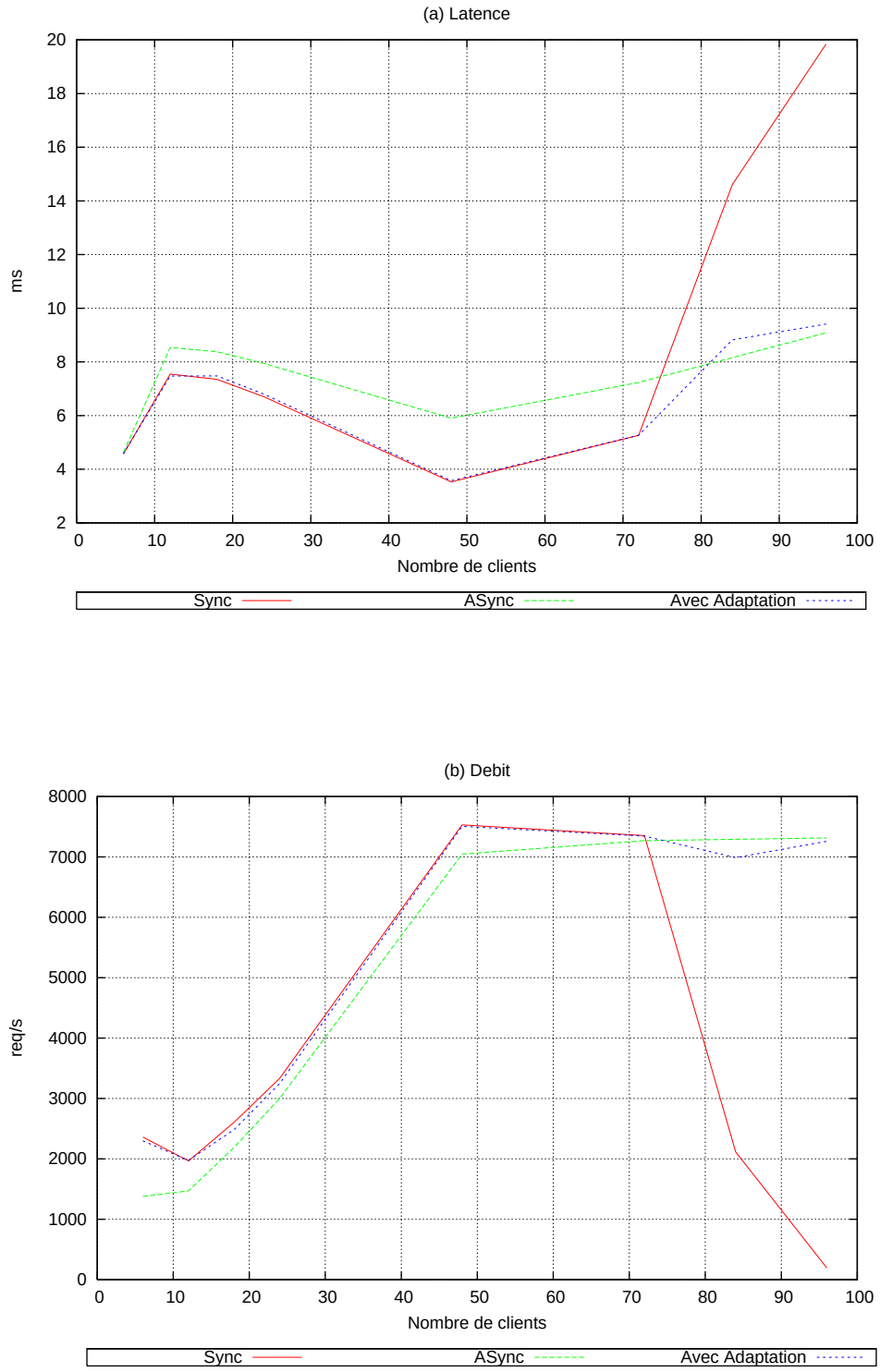

FIG. 4.3 – Comparaison du serveur dynamiquement adaptable par rapport aux résultats précédents dans le cas d'un serveur avec un goulot d'étranglement

Conclusion

Concevoir des applications performantes est une priorité depuis longtemps pour les développeurs. Plusieurs modèles de programmation ont donc été proposés au cours des trois dernière décennies. Les outils “historiques” sont la programmation à base de threads et celle à base d'événements. Nous avons détaillé, dans la partie consacrée à l'état de l'art, les avantages et les inconvénients de ces deux solutions. D'autres solutions essayant d'améliorer l'un ou l'autre de ces deux modèles de programmation ont également été proposées. La proposition la plus convaincante est la programmation à base d'étages qui permet de concevoir des applications performantes, résistantes à la charge.

Toutefois, ce type de programmation est lié à un modèle événementiel puisque la communication entre les étages est toujours effectuée de manière asynchrone. Comme nous l'avons expliqué, l'exécution des applications à étages gagne à être plus flexible, c'est-à-dire à supporter différents modes d'exécution.

Notre premier travail à donc été de proposer une solution basée sur les étages permettant de choisir entre différents modèles d'exécution. Cette solution, basée sur une implantation de SEDA, permet le passage à chaud d'un modèle d'exécution à un autre.

Nous avons ensuite cherché à déterminer dans quels cas une exécution synchrone est meilleure qu'une exécution asynchrone et inversement. Pour cela, nous avons effectué différents tests sur des serveurs de type serveurs web, en ne considérant que les cas totalement synchrone et totalement asynchrone. En comparant les résultats des exécutions synchrones et asynchrones, nous avons montré qu'il existe un gain réel, à la fois en termes de débit (jusqu'à 70%) et en termes de temps de réponse (jusqu'à 60%) à basse charge pour une exécution synchrone, alors qu'une exécution asynchrone se comporte mieux à haute charge. Ces expériences montrent qu'il est donc intéressant de pouvoir passer d'un modèle d'exécution à l'autre.

Nous avons enfin présenté une solution permettant le changement dynamique de modèle en fonction du débit sortant et de la charge du serveur. Les résultats ont montré que le coût de ce changement est négligeable et donc que les gains apportés par cette nouvelle architecture sont réels.

Perspectives

Dans ces travaux nous nous sommes essentiellement attachés à montrer qu'il existe un intérêt réel à un changement dynamique de modèle d'exécution. Les premiers résultats sont encourageants, mais sont limités au contexte applicatif des serveurs de données. De plus nous n'avons utilisé que des serveurs simplifiés. Ces résultats méritent donc d'être confirmés sur des serveurs de données complexes, ainsi que dans d'autres domaines applicatifs, comme par exemple celui des applications multimédia. Nous pensons d'ailleurs que sur d'autres applica-

tions, les gains peuvent être plus importants. En effet, le réseau ne masquera pas le surcoût engendré par une exécution asynchrone. Il sera également intéressant de regarder l'intérêt que peut avoir un modèle partiellement synchrone, ce que nous n'avons pas étudié dans ces travaux.

Actuellement, la prise de décision est basée sur des observations. Il va être important de voir comment, à partir des heuristiques que nous avons proposées (nombre de changements de contexte, taille de la file d'un étage, ...), il peut être possible de mettre en place une méthode plus générique (c'est à dire sans lien fort avec les spécificités d'une application). Dans ces travaux, nous ne nous sommes intéressés qu'à un serveur disposant d'un seul processeur. La présence ou non de plusieurs processeurs et leur influence est également un paramètre important à étudier. Dans une vue à plus long terme, il serait intéressant de regarder comment affecter les étages aux ressources d'exécution sous-jacentes pour tirer parti de ces dernières. Une bonne utilisation des caches ou encore des préoccupations de consommation d'énergie peuvent être des éléments à considérer.

Il est également intéressant d'étudier le comportement des applications avec différents injecteurs de charge. Dans ces travaux, nous nous sommes limités à l'étude des résultats avec un injecteur de charge en boucle fermée. L'étude du comportement avec des injecteurs de charge en boucle ouverte ou semi-ouverte fait partie des objectifs à court terme.

Enfin, un dernier objectif est de réaliser une implantation de SEDA avec le langage C, afin de disposer d'une deuxième plate-forme de test. En effet, le langage Java ne permet pas d'avoir un contrôle maximal sur les différents paramètres. Il est par exemple difficile d'avoir accès à certaines données tels que la manière dont les threads sont ordonnancés ou encore comment les threads de niveau utilisateur sont associés aux threads de niveau noyau. Différents choix ont été fait dans les bibliothèques et peuvent masquer certains comportements.

Bibliographie

[1] Haproxy, high performance tcp/http load balancer, http ://haproxy.1wt.eu/.

[2] Message passing interface, http ://www.mpi-forum.org/.

[3] Openmp, http ://www.openmp.org.

[4] Zeus web server, http ://www.zeus.com.

[5] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 289–302, Monterey, CA, June 2002. USENIX Association.

[6] Gustafsson Andreas. Threads without the pain. *ACM Queue*, 3(9) :34–41, 2005.

[7] Brendan Burns, Kevin Grimaldi, Alexander Kostadinov, Emery D. Berger, and Mark D. Corner. Flux : A language for programming high-performance servers. In *USENIX Annual Technical Conference*, pages 129–142, Boston, MA, May 2006. USENIX Association.

[8] Giorgio Calarco, Carla Raffaelli, Giovanni Schembra, and Giovanni Tusa. Comparative analysis of smp click scheduling techniques. In *International workshop on Quality of Service in Multiservice IP networks*, volume Volume 3375/2005, pages 379–389, Catania, Italy, February 2005. Springer Berlin / Heidelberg.

[9] Benjie Chen and Robert Morris. Flexible control of parallelism in a multiprocessor pc router. In *Proceedings of the General Track : 2002 USENIX Annual Technical Conference*, pages 333–346, Boston, Massachusetts, June 2001. USENIX Association.

[10] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazières, and Robert Morris. Event-driven programming for robust software. In *EW10 : Proceedings of the 10th ACM SIGOPS European workshop : Beyond the PC*, pages 186–189, Saint-Emilion France, September 2002. ACM Press.

[11] Richard P. Draves, Brian N. Bershad, Richard F. Rashid, and Randall W. Dean. Using continuations to implement thread management and communication in operating systems. In *SOSP '91 : Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 122–136, Pacific Grove California, United States, 1991. ACM Press.

[12] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *OSDI '99 : Proceedings of the third symposium on Operating systems design and implementation*, pages 101–115, New Orleans Louisiana, United States, 1999. USENIX Association.

[13] James R. Larus and Michael Parkes. Using cohort scheduling to enhance server performance (extended abstract). In *LCTES '01 : Proceedings of the ACM SIGPLAN workshop on Languages compilers and tools for embedded systems*, pages 182–187, New York NY USA, 2001. ACM Press.

[14] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2) :3–19, April 1979.

[15] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5) :33–42, May 2006.

[16] Frederic Mayot. Programmation parallèle à base de composants pour les applications de streaming, M2R S&L, Rapport M2R UJF, Septembre 2006.

[17] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *SOSP '99 : Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 217–231, Charleston South Carolina, United States, 1999. ACM Press.

[18] John K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference, January 1996.

[19] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash : An efficient and portable web server. In *USENIX Annual Technical Conference General Track*, pages 199–212, Monterey, California, USA, June 1999.

[20] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, , and Amol Shukla. Comparing the performance of web server architectures. In ACM, editor, *EuroSys'2007*, Lisboa, Portugal, March 2007.

[21] Li Peng and Zdancewic Steve. A language-based approach to unifying events and threads. Technical report, CIS Department, University of Pennsylvania, April 2006.

[22] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7) :54–62, 2005.

[23] Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03 : Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 19–24, Lihue, Hawaii, May 2003. USENIX Association.

[24] Robert von Behren, Jeremy Condit, Feng Zhou, George C. Necula, and Eric Brewer. Capriccio : scalable threads for internet services. In *SOSP '03 : Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, Bolton Landing, New York, USA, October 2003. ACM Press.

[25] Matt Welsh and David Culler. Adaptive overload control for busy internet servers. In *USENIX Symposium on Internet Technologies and Systems*, pages 4–4, Seattle, WA, USA, March 2003. USENIX Association.

[26] Matt Welsh, David Culler, and Eric Brewer. SEDA : An architecture for well-conditioned scalable internet services. In *SOSP '01 : Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, Banff Alberta Canada, 2001. ACM Press.

[27] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and M. Frans Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference General Track*, pages 239–252, San Antonio, Texas, June 2003. USENIX Association.

Résumé

Les performances constituent souvent un critère majeur pour les développeurs d'applications. De nombreux modèles de programmation ont été proposés pour les aider, tels que la programmation événementielle ou la programmation par threads. Différentes améliorations de ces modèles existent et notamment la programmation par étages. Ce modèle permet de concevoir des applications performantes et robustes à la charge. Cependant un de ses défauts est d'être fortement couplé à un modèle d'exécution. Dans cette étude, nous montrons qu'un découplage entre un programme et son modèle d'exécution permet de disposer d'applications plus robustes. Nous proposons une architecture à étages permettant une adaptation dynamique du modèle d'exécution en fonction de certaines heuristiques. Enfin, cette proposition est validée dans le contexte applicatif des serveurs de données.

Mots Clés : modèles d'exécution, modèles de programmation, programmation par étages, adaptation dynamique, serveurs de données