Implémentations efficaces de la concurrence sous Windows

Matthieu Boutier,
sous la direction de Juliusz Chroboczek,
Laboratoire PPS, Université Paris Diderot

Mars à août 2012

Table des matières

Introduction 3

1 Les mécanismes d'entrées/sorties asynchrones Windows 5
1.1 Entrées/sorties asynchrones sous Windows 5
1.2 Introduction aux jeux de tests 6
1.3 Serveurs réalisés 7
1.4 Problèmes rencontrés 7
1.5 Résultats 8
1.5.1 Résultats en moyenne 10
1.5.2 Résultats détaillés 10

2 Intégration à CPC 12
2.1 Restrictions 15
2.2 Jeux de tests et résultats 16
2.3 Hekate 17

Conclusion 18

A Les mécanismes d'entrées/sorties asynchrones Windows 20
A.1 `WaitForMultipleObjects` et `select` 21
A.2 Les *I/O Completion Ports (IOCP)* 22
A.3 Les *Completion Routines* 23
A.4 Autres méthodes 25

B Description des serveurs 26

C Résultats de tous les serveurs 27

D Comparatifs : IOCP contre file utilisateur 30

E Autres tests 32
E.1 *Sockets* dans l'état *Time Wait* et `SO_REUSEADDR` 32
E.2 Optimisations d'appels systèmes pour les entrées/sorties 32
E.3 L'appel système `select` et `FD_SETSIZE` 34

Introduction

La plupart des programmes sont concurrents, c'est-à-dire qu'ils doivent réaliser plusieurs actions en même temps. Par exemple, un client de messagerie peut récupérer les courriers pendant que l'utilisateur écrit un mail. Il y a deux façons de coder un programme concurrent : en utilisant des *threads* ou bien en utilisant un code à événements.

Threads Un *thread* est un flot de contrôle partageant sa mémoire avec tous les autres *threads* du processus. Un ordonnanceur s'occupe de répartir le temps d'exécution entre les différents *threads*. Ils peuvent être *coopératifs* ou *préemptifs*.

Un *thread* préemptif s'exécutera jusqu'à ce qu'il soit interrompu par son ordonnanceur, et n'a pas moyen de savoir quand il sera interrompu. Dans un programme concurrent à base de *threads* préemptifs, manipuler des variables communes à plusieurs *threads* nécessite donc des précautions : une méthode standard est d'utiliser des verrous, appelés *mutex*. Ceux-ci introduisent des risques d'inter-blocages.

A l'opposé, un *thread* coopératif s'exécutera jusqu'à ce qu'il décide de rendre la main *(coopérer)* à son ordonnanceur. Utiliser des *threads* coopératifs évite les problèmes de manipulation de ressources, puisque les *threads* ne sont jamais interrompus. En revanche, si un *thread* coopératif bloque, il bloquera tous les autres *threads*.

On parle souvent de *threads* natifs, pour désigner ceux fournis par le système d'exploitation, par opposition aux *threads* utilisateurs. Habituellement, les *threads* natifs sont préemptifs, et les *threads* utilisateurs coopératifs. Aussi, dans la suite de ce document, nous considèreront les *threads* natifs comme étant préemptifs.

Evénements La programmation par événements consiste à décomposer le flot de contrôle d'une tâche en plusieurs petites fonctions indépendantes, appelées *gestionnaires d'événements*. Ainsi découpé, le programme peut exécuter plusieurs flots de contrôle de manière concurrente, en exécutant tour à tour les gestionnaires d'événements leur correspondant.

La partie du code qui consiste à choisir et exécuter les gestionnaires d'événement s'appelle une *boucle à événements*. Elle opère à la manière d'un ordonnanceur coopératif : un gestionnaire d'événement sera exécuté de manière atomique.

Les gestionnaires d'événement doivent être relativement rapides à s'exécuter, pour garder le programme réactif. S'ils bloquent, toute la boucle à événements est bloquée : en particulier, il faut veiller à ne pas bloquer sur des entrées/sorties.

Concurrence et entrées/sorties Les programmes concurrents ont besoin de faire des entrées/sorties, que ce soit pour échanger des informations avec le disque, le réseau ou l'utilisateur. Le paradigme le plus courant est celui des entrées/sorties synchrones bloquantes : l'exécution courante est suspendue jusqu'à ce que les données soient lues ou écrites. Pour que le programme soit concurrent, il est alors

nécessaire d'utiliser plusieurs *threads* natifs : chaque *thread* peut bloquer sur une entrée ou une sortie, pendant que les autres s'exécutent.

Cependant, utiliser un grand nombre de *threads* peut avoir un coût significatif, et ne permet pas toujours d'atteindre au niveau de concurrence souhaité (nombre d'opérations s'effectuant en même temps). Notamment, créer un *thread* nécessite d'allouer sa pile, soit plusieurs dizaines de kilooctets [1].

Sous Unix, la technique habituelle pour faire plusieurs entrées/sorties concurrentes sur un seul *thread* est d'utiliser les entrées/sorties synchrones non-bloquantes. Lorsqu'un programme fait une entrée/sortie synchrone non-bloquante et que les données sont disponibles, les données sont copiées au programme, et son exécution continue. Dans le cas contraire, une erreur est retournée, mais son exécution continue aussi. Les entrées/sorties synchrones non-bloquantes ont été beaucoup étudiées, et montrées plus efficaces que les entrées/sorties synchrones bloquantes avec *threads* [GBSP04, KKK07, KC09].

Un autre modèle est celui des entrées/sorties asynchrones. Lorsqu'un programme fait une entrée/sortie asynchrone, le système enregistre la demande du programme, et rend l'exécution au programme. Celui-ci peut faire d'autres opérations en attendant que l'opération asynchrone se termine. Le système notifie le programme de l'achèvement de ces opérations. Ces notifications peuvent se faire de plusieurs façons.

Les entrées/sorties asynchrones sont peu utilisées et souvent difficilement utilisables sous Unix, bien que des recherches aient été faites [BMD99, LaH02]. Cependant, elles sont présentes depuis longtemps sous Windows. Ce système d'exploitation a développé au cours des années plusieurs méthodes pour réaliser les entrées/sorties asynchrones, tout en gardant la possibilité de faire des entrées/sorties synchrones (bloquantes et non-bloquantes) [2].

CPC Aussi bien les entrées/sorties non-bloquantes qu'asynchrones sont adaptées à une programmation en style à événements. L'expérience montre que programmer en style à événements est plus difficile qu'en style à *thread*, car le code en style à événements découpe le flot de contrôle en plusieurs gestionnaires d'événement indépendants, tandis que le code en style à *thread* le laisse visible au programmeur. Cependant, ce découpage est assez mécanique, et il est assez naturel de vouloir l'automatiser.

CPC est une extension du C développé par Chroboczek et Kerneis, qui permet d'écrire un code en style à *thread* [Chr06, CK10, KC11b]. Ce code est compilé en un code C équivalent en style à événements. CPC introduit la notion de *thread* CPC, un *thread* particulièrement léger pouvant s'exécuter, au choix du programmeur, dans une boucle à événements ou dans un *thread* natif. Il fournit des primitives de coopération permettant de ne pas bloquer. CPC a été pensé et conçu pour Unix, donc

1. La documentation Windows dit que la taille de la pile d'un *thread* est un multiple d'un nombre, typiquement 64 Ko, et que la taille par défaut est de 1Mo.

2. Cette évolution n'est pas figée, et la prochaine version de Windows annonce de nouvelles possibilités.

pour faire des entrées/sorties synchrones non-bloquantes. CPC fournit notamment une primitive de coopération adaptée à ce paradigme.

Enfin, le code produit par CPC a une efficacité proche d'un code à événements écrit à la main, comme l'ont démontré plusieurs tests [Ker08]. CPC a aussi fait ses preuves sur un vrai programme : Hekate. Hekate est un *seeder* BitTorrent très efficace réalisé en quelques mois seulement par des étudiants de M1 [AC09].

Terminologie Dans le monde Unix, on utilise les termes « descripteur de fichier » pour désigner une abstraction fournie par le système par laquelle le programme pourra communiquer avec ce que représente le descripteur de fichier. Celui-ci ne désigne pas nécessairement un fichier ; par exemple une *socket* est représentée par un descripteur de fichier. Dans le monde Windows, une abstraction analogue existe, sous la dénomination de *handle*. J'ai choisi d'utiliser ici la terminologie Unix : je parlerai donc de *descripteur de fichier*, ou plus simplement de *descripteur*.

Contributions Dans la première partie de ce document, nous explorons les différentes possibilités qui s'offrent au programmeur Windows, en présentant les méthodes principales pour faire des entrées/sorties asynchrones.

Dans la deuxième partie, nous montrons quelles sont leurs performances relatives, à travers les nombreux jeux de tests que j'ai réalisés.

Enfin, dans la troisième partie, nous étudions comment j'ai adapté CPC aux entrées/sorties asynchrones. Nous verrons comment j'ai porté Hekate sous Windows, quelles modifications ont été nécessaires, et quelles sont ses performances.

1 Les mécanismes d'entrées/sorties asynchrones Windows

Dans cette partie, je vais présenter de manière succincte les deux principales méthodes de faire des entrées/sorties asynchrones sous Windows, puis présenter les tests d'efficacité que j'ai réalisés.

Mes tests ne se limitent pas aux entrées/sorties asynchrones, mais utilisent aussi les entrées/sorties synchrones bloquantes, avec des *threads*, et les entrées/sorties synchrones non bloquantes, avec l'appel système `select`. Celui-ci vient du monde Unix, et est aussi présent sous Windows. Comme pour la version Unix, il prend en paramètre une liste de descripteurs de fichiers, et renvoie lesquels sont prêts pour une lecture ou une écriture.

1.1 Entrées/sorties asynchrones sous Windows

Les deux mécanismes principaux pour faire des entrées sorties asynchrones sous Windows sont les *I/O Completion Ports* et les *Completion Routines*. Le but de ces méthodes est de signaler au programme qu'une opération asynchrone s'est terminée afin qu'il puisse poursuivre son calcul. Je décris les différents mécanismes d'entrées/sorites asynchrones en détails dans l'annexe A.

Lors de tout appel asynchrone, l'utilisateur passe en paramètre de l'appel système une structure particulière qu'il a alloué : la structure `OVERLAPPED`. Le système se sert de cette structure tout au long de l'appel système asynchrone. Elle contient entre autre le statut de l'opération (le code d'erreur).

Les *I/O Completion Ports* Un *Input/Output Completion Port* (IOCP) est une file de messages : lorsque le système a terminé une opération asynchrone, il ajoute un message, appelé *paquet de complétion*, dans cette file. Le programme peut retirer les paquets de complétion de l'IOCP, et choisir la fonction à exécuter en conséquence. Un paquet de complétion est principalement constitué d'un pointeur vers la structure `OVERLAPPED` utilisée pour l'appel système.

Tous les appels systèmes asynchrones, hormis ceux utilisant les *completion routines*, peuvent s'utiliser avec les IOCP.

Les *Completion Routines* Les *completion routines* sont des fonctions exécutées à la fin de l'appel asynchrone. Elles sont définies par l'utilisateur, et données en paramètre aux appels systèmes utilisant les *completion routines*. Lorsque l'opération asynchrone sera terminée, la *completion routine* associée sera mise dans une file afin d'être appelée.

Les *completion routines* ne seront appelées que lorsque le programme sera dans un état d'attente alerte *(waiting and alertable state)*. Dans un tel état, le programme exécutera toutes les *completion routines*, jusqu'à ce que la file soit vide. Si, pendant l'exécution des fonctions de la file, de nouvelles *completion routines* sont ajoutées à la file, elles seront aussi appelées.

La signature des *completion routines* dépend de l'appel système, cependant, toutes prennent en argument le pointeur vers la structure `OVERLAPPED` ayant servi à l'appel. Peu d'appels systèmes permettent l'utilisation des *completion routines*, et on ne peut pas les utiliser avec des descripteurs de fichier utilisant les IOCP.

1.2 Introduction aux jeux de tests

Afin de mesurer l'efficacité des entrées/sorties sous Windows, j'ai implémenté un même serveur web minimaliste de plusieurs façons différentes. En effet, un serveur web doit pouvoir traiter les demandes de plusieurs milliers de clients simultanément. Il effectue donc de très nombreuses entrées/sorties, qui peuvent être partielles voire impossibles au moment de l'appel (les données entrantes ne sont pas prêtes, ou le tampon de sortie est plein). Ainsi, c'est une application idéale pour réaliser des tests de performances sur la concurrence dans un programme en général, et sur les entrées/sorties en particulier.

C'est une application que nous comprenons bien, même si elle demeure toujours difficile à mesurer, comme nous le verrons. Enfin, Kerneis et Chroboczek avaient utilisé cette application pour réaliser leurs tests [KC09], et il était naturel de continuer à leur suite.

Malgré la facilité apparente (connecter deux ordinateurs avec un câble, puis lancer un serveur d'un côté, un client de l'autre), et notre expertise locale en la matière, c'est un exercice qui est et demeure très difficile. En effet, beaucoup de problèmes dus aux différentes couches réseau peuvent apparaître, et les trouver peut demander beaucoup de temps.

Versions du système d'exploitation mesurées Les tests présentés ici ont été effectués sur Windows 7 professionnel, le dernier système d'exploitation Windows stable. Ils ont aussi été reproduits sur la version d'essai de Windows 8 (*Consumer preview*), mais exécutés sur une machine virtuelle ; les résultats sont sensiblement les mêmes.

1.3 Serveurs réalisés

J'ai écrit plusieurs serveurs web minimalistes, basés sur ceux que Kerneis a utilisés pour CPC [Ker08], mais utilisant différentes techniques de concurrence Windows. Leur implémentation combine les différentes API, la manière d'accepter les connexions, et la pré-allocation statique ou non des structures de données pour chaque client. Voici une liste des différents serveurs présentés :

- *completion routines* avec accept synchrone ;
- *completion routines* avec accept asynchrone (utilisant pour cela les IOCP) ;
- IOCP avec accept synchrone ;
- IOCP avec accept asynchrone et nombre de clients variable en fonction de la demande ;
- IOCP avec accept asynchrone et nombre de clients fixes ;
- `select` ;
- *threads* avec nombre de clients variables ;
- *threads* avec nombre de clients fixes ;

De plus, nous présentons un serveur écrit en CPC pour Windows, décrit en plus de détails dans la partie 2.2.

Tous ces serveurs, sauf ceux uniquement à base de *threads* (y compris CPC) sont écrits en style à événements, en utilisant un environnement contenant un pointeur de fonction indiquant quelle est la prochaine fonction à exécuter. Ils sont décrits plus en détail dans l'annexe B.

1.4 Problèmes rencontrés

Backlog windows Au débuts des tests, le serveur reçoit beaucoup de demandes de connexions en très peu de temps. Dans la plupart des cas, on observait que des paquets étaient réémis depuis le client : la connexion effective s'effectuait trois secondes plus tard que prévu.

Un problème similaire avait déjà été observé par Kerneis [Ker08], qui l'avait résolu en changeant la taille de la file d'`accept`, appelée *backlog*, afin qu'elle puisse accueillir toutes les connexions entrantes.

Sous Windows 7, la taille de cette file semble ne pas pouvoir être modifiée, et valoir 200. Cette limite apparait clairement sur certaines courbes : lorsque trop de requêtes arrivent en même temps, il est fréquent que des paquets soient perdus. Ils sont alors réémis trois secondes plus tard.

Une manière de limiter ce problème, parfois de l'éviter, est d'utiliser des `accept` asynchrones : en enregistrer assez dès le début permet souvent de ne pas perdre de paquets.

Limite de *sockets* utilisables atteinte Lors des certains tests, on pouvait voir jusqu'à 1000 connexions simultanées acceptées par le serveur, c'est-à-dire dont le client à reçu l'acquittement, mais le trafic réseau était nul.

Lors d'une fermeture de connexion TCP, celui des deux pairs qui a décidé la fermeture retiendra pour un certain temps l'adresse IP et le port de connexion de l'autre pair, afin de bloquer toutes les connexions venant de cette même adresse et de ce même port. La *socket* est alors dite dans l'état *Time Wait* [Pos81]. Cela évite d'avoir un paquet lent de la connexion précédente qui vienne perturber la nouvelle connexion. Cependant, dans le cas d'un serveur web, c'est le serveur qui ferme la connexion, lorsqu'il a fini d'envoyer le fichier [FTY99]. Etant donné que nous réalisons nos tests avec un seul client, l'adresse IP ne change pas, et très vite, tous les numéros de ports sont atteints.

Windows retient ces adresses pendant 120 secondes. J'ai réalisé plusieurs tests essayant d'obtenir de meilleurs résultats, sans succès. Plus de détails sont disponibles dans l'annexe E.1.

Perte de niveau de concurrence On observe sur certains serveurs que le niveau de concurrence demandé par le client diminue au cours du temps, en général lorsque des requêtes sont perdues. Il s'agit probablement d'un bug d'apacheBench, mais je n'en ai pas trouvé l'origine.

1.5 Résultats

Dans cette partie, nous présentons la comparaison des performances des différents serveurs. Nous montrons d'abord les caractéristiques en moyenne des différents serveurs, permettant de les comparer rapidement et efficacement, puis nous considérons des détails sur une sélection de serveurs. Chaque test présenté ici est réalisé avec 50 000 requêtes.

Conditions expérimentales Les tests ont été réalisés avec un serveur Intel Pentium M à 1,70 Ghz, 1Go de mémoire vive, et une interface *Ethernet* à 100 Mb/s. Il s'exécute sous Windows 7 Professionnel. Le client est un AMD Athlon 64 3500+ à 2,20 GHz, 1Go de mémoire RAM. Il s'exécute sous Debian, version 6.0.3. Le client teste le serveur avec apacheBench, un logiciel fourni avec le serveur web Apache permettant de tester les performances d'un serveur web. Toutes les mesures réalisées se font côté client.

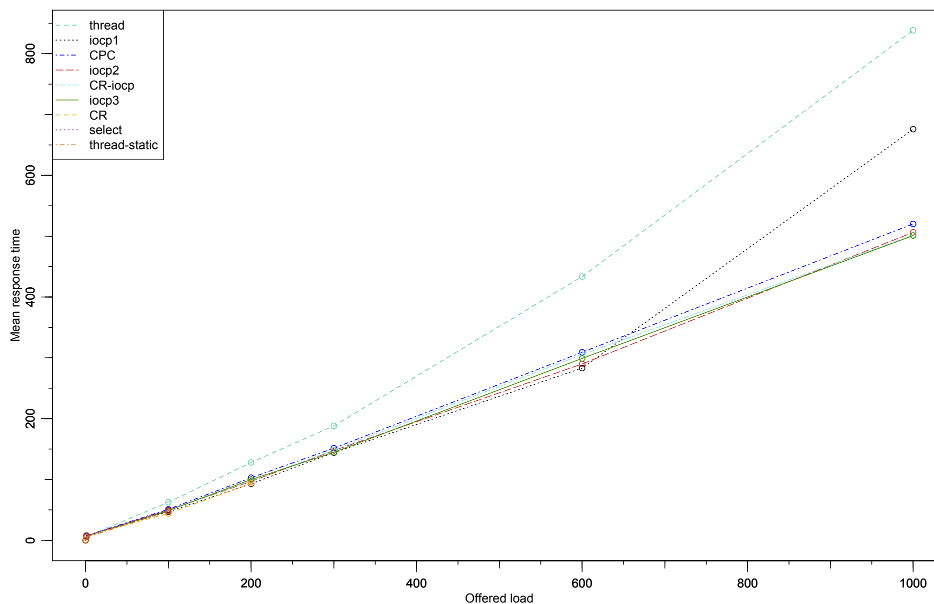

FIGURE 1 – Comparaison des performances en fonction de la charge

Calcul des résultats en moyenne Il ne suffit pas de prendre la moyenne du temps d'exécution des requêtes, ni leur médiane, ni de diviser le nombre de requêtes par le temps total d'exécution pour avoir des résultats significatifs de ce qu'on veut mesurer. En effet, plusieurs phénomènes apparaissant dans les résultats détaillés montrent que les résultats pourraient en être faussés.

Au début de l'expérience, beaucoup de requêtes sont envoyées au serveur, et il est fréquent qu'il n'arrive pas à toutes les accepter. Certaines sont alors perdues, et le niveau de concurrence effectif du serveur n'est pas celui demandé. Les requêtes perdues sont réémises par le client au bout de trois secondes. Souvent, le serveur devient alors capable de gérer autant de requêtes qu'est le niveau de concurrence, en même temps. Par ailleurs, certains serveurs ne sont pas capables d'assumer la charge demandée. Le temps de réponse à chaque requête n'est donc pas significatif d'un temps de réponse que le serveur mettrait à la charge demandée.

Enfin, Certains serveurs donnent lieu à des erreurs, qui sont difficilement détectables dans les mesures récupérées. Parfois, celles-ci sont incohérentes (temps d'attente supérieur au temps total) : on peut ainsi les ignorer.

Je n'ai donc retenu que les requêtes s'effectuant lorsque la charge atteinte par le serveur satisfait la charge demandée, et j'ai ignoré les requêtes réémises, ainsi que celles dont j'ai pu détecter qu'elles provenaient d'une erreur.

1.5.1 Résultats en moyenne

La figure 1 montre des courbes représentant pour chaque serveur le temps de réponse moyen en fonction de la charge. Une courbe qui s'arrête prématurément signifie que le serveur n'est pas capable d'assumer la charge demandée. Les résultats du serveur écrit en CPC sont commentés dans la partie 2.2.

Nous voyons que seuls six serveurs ont réussi à atteindre le niveau de concurrence demandé : les quatre serveurs utilisant des `accept` asynchrones (CR-iocp, IOCP2, IOCP3 et CPC), le serveur avec IOCP utilisant des `accept` synchrones, et le serveur allouant un *thread* par connexion. Les quatre serveurs avec `accept` asynchrones ont des progressions linéaires et les meilleurs résultats (réponse moyenne d'environ 500ms pour 1000 requêtes simultanées). Les deux autres serveurs perdent leur progression linéaire lorsque la charge augmente trop, et sont plus lents (700 à 800ms). Quant aux autres serveurs, ils n'arrivent pas à assumer des charges importantes.

Par ailleurs, apacheBench, le client réalisant les tests, relève que seuls les serveurs à base de `accept` asynchrones ne font pas d'erreurs. Voici le nombre d'erreurs relevées à un niveau de concurrence de 1000 :

CPC	0	iocp1	1625	select	13562
CR	19884	iocp2	0	thread	18608
CR-iocp	0	iocp3	0	thread-static	16247

Les seuls serveurs à tenir la charge sans erreurs sont donc ceux utilisant les `accept` asynchrones. Parmi eux, l'utilisation des *completion routines* ou des IOCP donne lieu à des résultats similaires. Le serveur utilisant les entrées/sorties non-bloquantes, avec `select`, ne tient pas 200 connexions simultanées. De plus, le nombre d'erreurs reportées est très élevé.

1.5.2 Résultats détaillés

Les résultats précédents sont suffisants pour une première approche, mais ne permettent pas d'étudier des comportements plus précis, ni de comprendre pourquoi ces résultats sont produits : il est nécessaire de surveiller le comportement de chaque requête au fil du temps. J'ai sélectionné quatre serveurs, significatifs des différents comportements observés. Les courbes correspondant aux autres serveurs sont disponibles en annexe.

J'ai choisi les courbes de quatre serveurs avec une valeur de concurrence de 1000 : ceux utilisant les *completion routines* (CR), les IOCP avec `accept` asynchrone et nombre de connexions dynamiques, `select`, et les *threads* avec création d'un *thread* lors de l'acception d'une connexion.

Représentations Sur chaque graphique, deux données sont représentées, où l'abscisse représente le temps depuis le début de l'expérience. La première est un nuage de points. Chaque point représente l'accomplissement d'une requête. En ordonnée

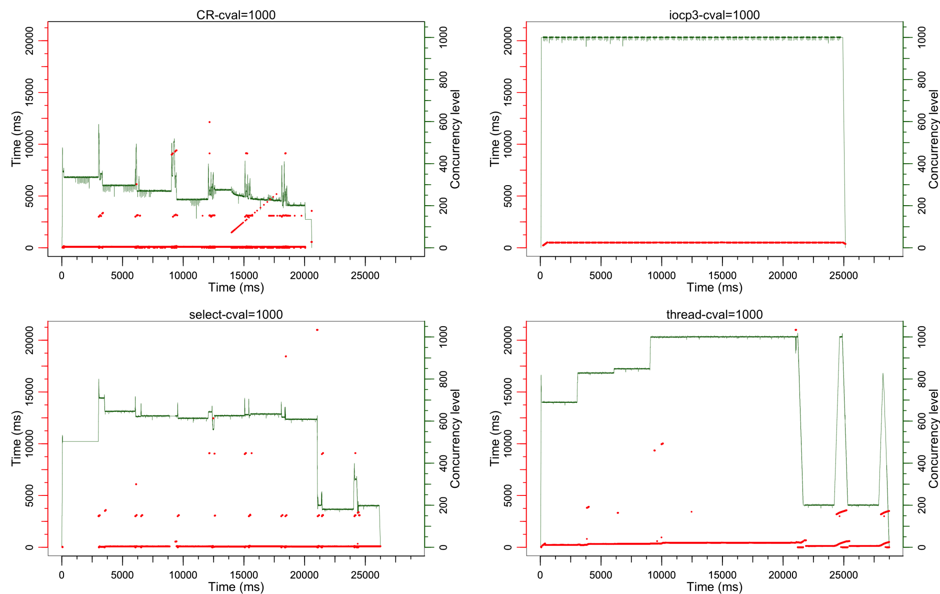

FIGURE 2 – Jeu de tests de serveurs Web

(échelle de gauche) se trouve le temps qu'a mis la requête avant d'être complétée. La deuxième est le niveau de concurrence côté serveur en fonction du temps, c'est-à-dire le nombre de connexions que le serveur traite simultanément.

Interprétation Les résultats obtenus sont assez significatifs des différentes méthodes, mais malheureusement sont assez instables : certaines courbes laissent apparaitre, pour le même serveur et dans les mêmes conditions, de meilleurs résultats que d'autres, et je n'ai pas réussi à les stabiliser. Ces différences rentrent dans le cadre des problèmes décrits dans la partie 1.4 : perte ou non de paquets, attente inactive du serveur.

La figure 3 représente les résultats des 4 serveurs sélectionnés pour cette partie. On observe que seul le serveur IOCP3 arrive à tenir la charge (courbe du haut), et son comportement est uniforme tout au long de l'exécution. Il ne perd aucun paquet tout au long de celle-ci. Les trois autres en revanche offrent de mauvais résultats. Celui utilisant des *completion routines* est particulièrement déplorable, avec un niveau de concurrence atteint relativement faible et beaucoup de paquets perdus. Le serveur utilisant `select`, avec les entrées/sorties synchrones non-bloquantes, perd aussi beaucoup de paquets et n'arrive pas à supporter la charge. Par ailleurs, le client relève un nombre particulièrement élevé d'erreurs pour ce serveur. Enfin, le serveur à base de *threads* arrive progressivement jusqu'à 1000 connexions simultanées, mais au bout d'un temps très long, et au prix de plusieurs paquets perdus. Il n'est par ailleurs pas capable de maintenir ce niveau au cours du temps de manière fiable.

Le serveur utilisant les IOCP est donc de loin le meilleur des ceux présentés, tant en fiabilité qu'en efficacité. S'il lui arrive, comme tous les autres serveurs, de perdre des paquets au début d'un test, il en perd généralement moins que les autres, et il lui arrive, comme c'est le cas ici, de n'en perdre aucun.

2 Intégration à CPC

CPC est une extension du langage C qui permet de créer des *threads* particulièrement légers [Ker08][KC11a][KC11b].

Un programme CPC en style à *thread* est compilé vers un programme C équivalent en style à événements. CPC peut s'intégrer à plusieurs boucles à événements, du moment qu'elles implémentent les primitives CPC.

Quand les *threads* CPC sont créés, ils sont coopératifs, et s'exécutent dans la boucle à événements utilisant les entrées/sorties non-bloquantes. Ils peuvent la quitter pour s'exécuter de manière préemptive dans une *threadpool*, avec la primitive `cpc_detach`. Le *thread* CPC est alors dit *détaché*.

Les *threads* CPC, lorsqu'ils sont coopératifs, s'exécutent jusqu'à ce qu'ils coopèrent, en utilisant les primitives de coopération du langage. Ils ne doivent pas bloquer entre deux points de coopération, sans quoi toute la boucle à événements se retrouve bloquée, et avec elle tous les *threads* coopératifs. En particulier, ils ne doivent pas bloquer sur des entées/sorties.

Chroboczek et Kerneis ont écrit une boucle à événements et une *threadpool* pour la version Unix de CPC. Je les ai repensées et implémentées afin qu'elles s'adaptent aux entrées/sorties asynchrones utilisant les IOCP. Nous décrivons ici dans un premier temps la primitive du CPC d'origine, puis exposons la nécessité d'une nouvelle primitive, la présentons, et en donnons des détails d'implémentation. Enfin, nous montrons que l'implémentation faite est efficace.

La primitive cpc_io_wait CPC a été conçu pour Unix, et utilise donc les entrées/sorties synchrones non-bloquantes. Pour coopérer, CPC introduit une primitive de synchronisation, `cpc_io_wait` [3], qui peut précéder un appel système : le *thread* CPC sera suspendu jusqu'à ce que le descripteur de fichier spécifié en paramètre soit prêt pour une lecture/écriture. Pendant ce temps, les autres *threads* CPC continuent à s'exécuter.

La propriété importante de `cpc_io_wait` est de s'appliquer à n'importe quel descripteur de fichier, indépendamment de l'opération désirée : le programmeur n'est pas limité à un jeu de fonctions déjà implémentées dans le *runtime*. La librairie CPC, contenant des fonctions telles `cpc_read` et `cpc_write` [4], est entièrement écrite en CPC pur : ce ne sont pas des primitives. On peut donc étendre cette librairie, sans restrictions, et sans modifier le *runtime*.

3. `cps int cpc_io_wait(int fd, int direction, cpc_condvar *cond)`

4. `cps int cpc_read(int fd, void *buf, size_t count)`
`cps int cpc_write(int fd, void *buf, size_t count)`

Il est utile de pouvoir annuler une opération en attente, ce qui se fait dans CPC avec les *variables de condition*. Les variables de conditions sont des objets fournis par CPC sur lesquels un ou plusieurs *threads* CPC peuvent se mettre en attente. Un *thread* CPC peut réveiller un autre *thread* en attente sur une variable de condition en signalisant cette dernière. Les primitives CPC permettant d'attendre, dont `cpc_io_wait`, prennent en argument une variable de condition qui, si elle est signalée, interrompt l'appel : la primitive retourne avec une erreur.

La nouvelle primitive Lors d'une entrée/sortie asynchrone, le système enregistre tout de suite l'opération à faire, sans attendre que le descripteur soit prêt. Il n'est donc pas possible d'implémenter `cpc_io_wait` en termes d'entrées/sorties asynchrones [5].

La solution adoptée est une primitive prenant un pointeur de fonction en paramètre. Cette fonction, implémentée par l'utilisateur, effectuera l'appel asynchrone ; son type est défini par :

```
typedef int64_t (*cpc_async_prim)(HANDLE h, void *closure,
                                  OVERLAPPED *ovl);
```

Ses trois arguments sont respectivement le descripteur de fichier `h` sur lequel sera faite l'opération asynchrone, une fermeture `closure` servant à transmettre des données (par exemple le déplacement où lire dans un fichier), et un pointeur `ovl` vers une structure `OVERLAPPED`.

La structure `OVERLAPPED` sur laquelle pointe `ovl` est allouée par le *runtime*, qui exécutera la fonction lors d'un appel à la primitive. Celle-ci a pour prototype :

```
cps int64_t
cpc_call_async_prim(HANDLE handle, cpc_async_prim f,
                    void *closure, cpc_condvar *cond);
```

où `handle` est le descripteur sur lequel on effectue l'opération asynchrone, `f` la fonction qui fera l'appel asynchrone, `closure` la fermeture à repasser à `f`, et `cond` une variable de condition associée à l'opération.

La primitive `cpc_call_async_prim` va appeler la fonction `f`, en lui passant notamment une structure `OVERLAPPED` qu'elle aura allouée. La fonction `f` retourne l'erreur `ERROR_IO_PENDING` (ou `WSA_IO_PENDING`) pour indiquer qu'une opération asynchrone est en cours. La primitive coopère alors avec la boucle à événements, en se retirant de la file de *threads* à exécuter. Si la fonction `f` indique qu'il n'y a pas d'erreur (en retournant une valeur positive), alors la primitive rend immédiatement l'exécution au *thread*.

5. Dans le cas particulier de Windows, on peut attendre qu'un descripteur de fichier soit prêt en lecture/écriture avec un appel asynchrone. Il suffit d'opérer sur 0 octets. Par exemple, lire 0 octets avec `ReadFile` de manière asynchrone sur une *socket* ne produira pas de paquet de complétion, jusqu'à ce que la *socket* aie des données à lire. Cependant, ce comportement n'est pas documenté, ne reflète pas les entrées/sorties asynchrones et ne permet pas de les exploiter.

Annulation d'une opération asynchrone On veut pouvoir annuler une opération asynchrone, de même qu'on peut interrompre un *thread* en attente sur `cpc_io_wait`. Dans la version Unix de CPC, cela se faisait en signalant la variable de condition passée à la fonction `cpc_io_wait` : j'ai voulu garder le même mécanisme.

Sous Windows, pour annuler une opération asynchrone, deux informations sont nécessaires : un pointeur vers la structure `OVERLAPPED` associée à l'opération, et le descripteur sur lequel l'opération s'exécute. La fonction `cpc_call_async_prim` alloue la structure `OVERLAPPED`, donc en connait l'adresse, et reçoit en argument le descripteur de fichier sur lequel l'opération est effectuée, ainsi que la variable de condition associée à l'appel. Le *runtime* retient ces informations jusqu'à la fin de l'appel asynchrone : si la variable de condition est signalée, il pourra interrompre celui-ci.

Modifications de la boucle à événements La boucle à événements d'origine de CPC se déroule en quatre étapes : l'exécution des *threads* CPC prêts, puis de ceux qui étaient endormis et pour qui il était temps de se réveiller, puis le rapatriement des *threads* CPC détachés, et enfin le traitement de ceux qui sont en attente d'entrées/sorties, à l'aide de l'appel système `select`.

Pour cette dernière étape, `select` est appelé avec, en argument, l'ensemble des descripteurs pour lesquels les *threads* CPC sont en attente. Lorsque `select` retourne, l'ensemble passé en argument contient alors tous les descripteurs prêts. Le *runtime* CPC parcourt cette liste, et retrouve pour chaque descripteur un *thread* en attente sur ce descripteur, pour qu'il reprenne son exécution.

Dans l'adaptation windows, les deux premières étapes sont conservées, mais les deux dernières fusionnées et simplifiées. En effet, elles sont toutes deux remplacées par une boucle récupérant tous les paquets de complétion présents dans l'IOCP.

La clef de chaque paquet est analysée par cas, pour déterminer la nature du paquet : paquet de complétion d'entrée/sortie, ou paquet posté par le *runtime* (voir paragraphe suivant).

Lorsqu'il s'agit d'un paquet de complétion d'entrée/sortie, le pointeur vers la structure `OVERLAPPED` indique un offset déterminé dans la structure du *thread* CPC, dont on récupère l'adresse. Le *thread* CPC est alors remis dans la file des *threads* à exécuter : il le sera au prochain tour de boucle.

Modification du mécanisme de *threadpool* Une *threadpool* est un ensemble de *threads* natifs qui se partagent un travail. Dans le cas de CPC, ce travail consiste à exécuter des *threads* CPC de manière préemptive. Un *thread* CPC détaché dans une *threadpool* peut rejoindre la boucle à événements pour y continuer son exécution ou pour y mourir (il s'est terminé dans la *threadpool*, mais doit prévenir la boucle principale qu'il termine). Il peut aussi vouloir créer un *thread* CPC : rappelons qu'un *thread* CPC nouvellement créé est toujours rattaché à la boucle à événements.

Dans la version Unix, les *threads* CPC exécutés dans la *threadpool* peuvent se rattacher à la boucle à événements en se mettant en attente dans une file particulière

propre à la *threadpool*. La boucle événements, à chaque tour, récupèrera les *threads* CPC alors présents dans cette file, et les rattachera à elle.

Dans la version Windows, nous nous servons de la file des IOCP à la place de la file de la *threadpool*. La boule à événements rattachera les *threads* CPC issus de la *threadpool* en même temps que les paquets de complétion d'appels asynchrones. Afin de différencier les différents paquets, trois clefs supplémentaires sont réservées, déterminant la nature de l'opération de rattachement.

Détails d'implémentation Lorsqu'une entrée/sortie asynchrone utilisant les IOCP est terminée, on ne récupère qu'un pointeur vers la structure `OVERLAPPED` donnée au système lors de l'appel asynchrone.

Pour déterminer quel *thread* CPC a émis l'appel asynchrone à partir d'un pointeur sur la structure `OVERLAPPED` liée à l'appel, deux solutions s'offrent à nous. La première est de disposer d'une table d'association dont la clef est l'adresse de la structure `OVERLAPPED`, et la valeur est l'adresse de la structure du *thread* CPC correspondant. La seconde est d'encapsuler la structure `OVERLAPPED` dans une structure plus grande. Il est alors possible de retrouver la structure plus grande, et avec elle toutes les informations ajoutées. Il est notamment possible de l'inclure dans la structure du *thread* CPC lui-même.

J'ai choisi d'utiliser la deuxième méthode dans mon implémentation du *runtime* CPC, en incluant la structure `OVERLAPPED` dans la structure du *thread* CPC. Cette solution permet d'obtenir l'adresse du *thread* en O(1), mais c'est au *runtime* d'allouer la structure `OVERLAPPED`, et non à l'utilisateur. C'est pourquoi nous avons besoin de donner la fonction à appeler depuis le runtime.

2.1 Restrictions

Certaines techniques de programmation utilisées avec la version Unix de CPC dépendent de la nature synchrone des entrées/sorties. Ces techniques ne sont donc pas applicables avec la version Windows de CPC.

Allocation retardée L'allocation retardée consiste, dans le cas d'une lecture synchrone, à allouer le tampon servant à récupérer les données seulement lorsqu'elles sont prêtes. Dans le cas asynchrone, cette optimisation n'est pas possible : le tampon doit être alloué au moment de l'appel, même s'il n'y a pas de données prêtes.

La technique de l'allocation retardée limite le nombre de tampons mémoire utilisés simultanément. Par exemple, si un serveur web a beaucoup de clients lents, il peut n'avoir que très peu de tampons alloués à la fois.

Partage mémoire Le partage mémoire consiste à n'avoir qu'un seul tampon pour plusieurs lectures ou écritures. Cela requiert de ne se servir du tampon que pour une tâche en même temps.

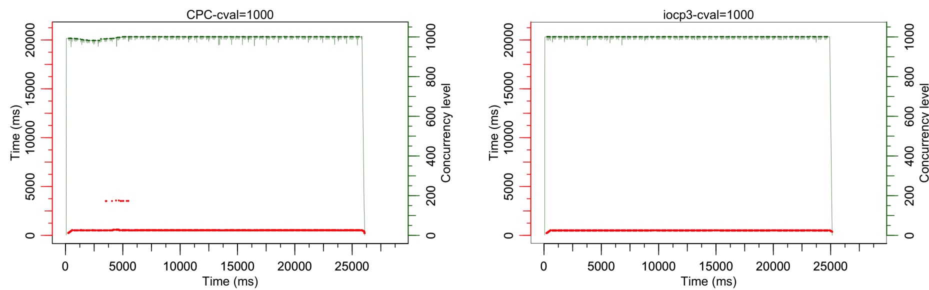

FIGURE 3 – Benchmarks IOCP vs CPC

Cette optimisation n'est impossible que dans le cas asynchrone, car du parallélisme apparaît au niveau des entrées/sorties. En effet, si deux descripteurs sont prêts en même temps, les deux opérations asynchrones vont s'effectuer en parallèle. Il est donc absolument nécessaire que les tampons mémoire soient distincts pour toutes les opérations asynchrones s'effectuant en même temps.

2.2 Jeux de tests et résultats

Comme je l'ai déjà mentionné dans la partie 1.2, les performances de CPC ont été testées sous Unix avec, notamment, un serveur Web écrit en CPC. L'implémentation du serveur CPC s'adapte à la version Windows avec peu de modifications, car les entrées/sorties utilisent la librairie CPC, qui a été refaite en conséquence. La seule modification de fond concerne la manière d'accepter les connexions entrantes.

Dans la version Unix, un *thread* CPC boucle en faisant autant de `accept` synchrones non-bloquants que possible, et en créant un *thread* pour chaque connexion reçue. Dans la version Windows, on utilise des `accept` asynchrones : afin de pouvoir toujours accepter une connexion dès qu'elle arrive, il faut qu'au moins un `accept` asynchrone soit en cours. J'ai donc choisi, comme pour les serveurs en C pur, d'initialiser plusieurs `accept` asynchrones au début du programme. Dans CPC, cela consiste simplement en une boucle qui crée autant de *threads* CPC que de connexions désirées.

Les résultats en moyenne exposés dans la figure 1 de la partie 1.5.1 montrent que le serveur écrit en CPC est très compétitif, car à peine plus lent (quelques millisecondes) que les meilleurs serveurs écrits à la main.

La figure 3 compare le serveur écrit en CPC avec le meilleur des serveurs écrit à la main. Les résultats montrent que le serveur écrit en CPC supporte la montée de charge sans problème. Sur cette figure, on peut voir quelques requêtes particulièrement lentes vers le début du test, dans le cas de CPC. Il s'agit de requêtes perdues et réémises, et non d'un ralentissement du serveur. Ces quelques cas pathologiques n'arrivent pas toujours, et ne sont pas propres à CPC dans le cas

général : il arrive aussi que le serveur écrit à la main perde des requêtes.

Ces tests montrent que CPC fonctionne parfaitement sous Windows, et qu'il est compétitif avec du code à événement écrit à la main.

2.3 Hekate

Hekate est un *seeder* bit-torrent écrit en CPC [AC09] lors d'un stage par des étudiants de M1. C'est le premier programme conséquent écrit en CPC. Il ne comprend que 3500 lignes de code, et Kerneis et Chroboczek l'estiment compétitif avec des programmes bien plus gros. L'implémentation de Hekate a montré que CPC fonctionnait pour des programmes réalistes non triviaux.

Changements apportés La version Unix de CPC utilise les entrées/sorties non-bloquantes. Cependant, celles-ci se comportent comme des entrées/sorties bloquantes lorsqu'elles opèrent sur disque.

Afin de lire un fichier sur disque sans bloquer, la version Unix de Hekate appelle la fonction `memmap` pour mettre le fichier en mémoire, et s'assure auprès du système que les données sont dans le cache. Si elles n'y sont pas, le *thread* rend l'exécution à la boucle principale, en espérant qu'elles y seront au prochain tour de boucle. Si elles n'y sont toujours pas, alors le *thread* se détache dans un *thread* natif pour s'exécuter sans bloquer.

Cette optimisation importante et complexe devient inutile lorsqu'on utilise des entrées/sorties asynchrones : lorsqu'une lecture asynchrone se termine, les données sont dans le tampon passé à l'appel système. Le *thread* n'a donc pas eu à bloquer, et dispose des données.

Hekate utilise aussi les techniques d'allocation retardée, et de partage de tampons. Ces méthodes, comme nous l'avons vu précédemment (Paragraphe 2.1), ne sont pas possibles avec des entrées/sorties asynchrones.

Résultats Pour porter Hekate sous Windows, j'ai du implémenter toutes les fonctions de la librairie standard de CPC, et en ajouter. Celles-ci utilisent la primitive que j'ai définie, vérifiant son bon fonctionnement.

L'implémentation actuelle de Hekate sur Windows fonctionne parfaitement, et ne perd pas la légèreté ni l'efficacité de la version Unix. Nous l'avons testé sur un réseau de 100Mb/s : Hekate sature ce réseau, avec une faible utilisation de CPU.

Chroboczek soutient qu'il s'agit de la deuxième implémentation efficace de BitTorrent sous Windows, avec μTorrent.

Avoir pu implémenter Hekate avec si peu de modifications signifie qu'il aurait pu être de base écrit par les mêmes étudiants de M1 qui l'ont écrit sous Unix. Cela signifie que la programmation utilisant les IOCP, réputée difficile, est maintenant accessible à un moindre coût par l'utilisation de CPC.

Conclusion

J'ai écrit et effectué de nombreux tests d'efficacité des primitives d'entrées/sorties de Windows. Ces tests sont documentés, et peuvent être facilement reproduits. La suite de résultats obtenue est, à notre connaissance, unique dans la littérature. Elle montre que pour être efficace sous Windows, il faut utiliser des entrées/sorties asynchrones avec les IOCP, conformément à ce qui se dit dans la communauté Windows.

J'ai porté CPC sous Windows, en le modifiant pour qu'il puisse utilise les IOCP. La traduction du code CPC en code C demeure inchangée, car elle s'adapte bien à l'asynchrone. En revanche, le *runtime* était conçu pour des entrées/sorties non-bloquantes. J'ai été amené à concevoir une nouvelle primitive de synchronisation, adaptée aux entrées/sorties asynchrones, et respectueuse de la sémantique de CPC. Cette primitive, j'en suis convaincu, peut s'étendre à d'autres langages qu'à CPC. J'ai implémenté et effectué des tests utilisant la version Windows de CPC. Les résultats montrent qu'un programme CPC a des performances très semblables à un code C en style à événement écrit à la main.

Enfin, j'ai porté Hekate sous Windows. Hekate est un *seeder* BitTorrent écrit pour la version Unix de CPC. Le porter sous Windows a montré que certaines techniques d'optimisation étaient liées à l'utilisation d'entrées/sorties synchrones : certaines ne sont plus nécessaires, d'autres ne peuvent plus être faites. La version Windows de Hekate fonctionne parfaitement, et Chroboczek soutient qu'il s'agit de la deuxième implémentation efficace de BitTorrent sous Windows, avec μTorrent. Enfin, l'implémentation de Hekate montre que la programmation asynchrone, réputée difficile, est maintenant accessible à moindre coût grâce à CPC.

Remerciements

Je tiens à remercier Juliusz Chroboczek pour son encadrement attentif, patient et compréhensif, Greg Hazel pour l'idée originale de ce stage, son partage d'expérience en programmation Windows, et ses commentaires sur les résultats obtenus, et Gabriel Kerneis pour son aide, son soutient constant et son savoir sur les jeux de tests, les serveurs Web et CPC. Merci aussi à ceux qui m'ont écouté parler de mes résultats, en particulier Grégoire Henry et Flavien Breuvart.

Références

[AC09] P. Attar et Y. Canal. Réalisation d'un seeder bittorrent en CPC. Rapport de stage de M1, PPS, Université Paris 7, 2009.

[BMD99] Gaurav Banga, Jeffrey C. Mogul, et Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '99, Berkeley, CA, USA, 1999. USENIX Association.

[Chr06] Juliusz Chroboczek. Continuation-passing for C : a space-efficient implementation of concurrency. Technical report, PPS, Université Paris 7, 2006.

[CK10] J. Chroboczek et G. Kerneis. *The CPC manual*, 2010.

[FTY99] T. Faber, J. Touch, et W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1573–1583. IEEE, 1999.

[GBSP04] L. Gammo, T. Brecht, A. Shukla, et D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Ottawa Linux Symposium*, volume 19, 2004.

[KC09] Gabriel Kerneis and Juliusz Chroboczek. Are events fast? Technical report, PPS, Université Paris 7, January 2009. 5 pages.

[KC11a] Gabriel Kerneis and Juliusz Chroboczek. Continuation-Passing C, compiling threads to events through continuations. *Higher-Order and Symbolic Computation*, 24(3) :239–279, 2011.

[KC11b] Gabriel Kerneis and Juliusz Chroboczek. CPC : programming with a massive number of lightweight threads. In *Proceedings of the Fourth Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, pages pp. 30–34, Saarbrücken, Allemagne, April 2011.

[Ker08] Kerneis. CPC, des threads coopératifs par passage de continuations. Master's thesis, Université Paris 7, 2008.

[KKK07] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, Berkeley, CA, USA, 2007. USENIX Association.

[LaH02] B.C.R. LaHaise. An AIO Implementation and its Behaviour. In *Ottawa Linux Symposium*, page 260, 2002.

[Pos81] J. Postel. RFC 793 : Transmission Control Protocol. *Status : Standard*, september 1981.

[RN11] J. Richter and C. Nasarre. *Windows via C/C++*. Microsoft Press, 2011.

A Les mécanismes d'entrées/sorties asynchrones Windows

Dans cette partie, nous explorons les fonctions et structures de données que Windows met à notre disposition pour faire des entrées/sorties asynchrones. Nous les comparons dans la partie suivante.

API Windows L'API Windows fournit une pléthore d'appels systèmes, pour de nombreuses manières de faire des entrées/sorties, synchrones et asynchrones. Certaines de ces techniques fonctionnent en attendant sur un ensemble d'objets, d'autres en ajoutant des notifications de complétion dans une file, d'autres encore en exécutant une fonction (*callback* ou continuation) lorsque l'opération se termine. Ces méthodes sont parfois incompatibles entre-elles, mais parfois elles se complètent [6].

Nous n'avons trouvé aucun article de recherche mesurant l'efficacité des différentes méthodes ; il nous importait donc de comprendre chacun des mécanismes, et de les comparer à travers plusieurs jeux de tests.

La documentation Windows ainsi que beaucoup de sites internet et de *blogs* affirment que la méthode la plus efficace est d'utiliser les entrées/sorties asynchrones avec les *Input/Output Completion Port* (IOCP). Richter et Nasarre [RN11], considérés comme une référence de la programmation pour Windows, recommandent aussi l'utilisation des IOCP.

Les événements noyau Les *événements noyau (kernel Events)* sont des objets alloués par le système, potentiellement partagés entre tous les programmes. Un programme peut se mettre en attente sur un événement noyau. Il sort de son attente lorsque l'événement noyau est signalisé. Il peut être signalé par un autre programme, ou un autre *thread* du même programme, ou encore par le système, par exemple à la fin d'une opération asynchrone. S'il y a plusieurs *threads* en attente sur un même événement noyau, un d'entre eux sera signalisé.

Les événements noyau sont donc semblables aux variables de conditions, à quelques différences près. Premièrement, si aucun *thread* n'est en attente sur l'événement noyau, il restera signalé. Deuxièmement, les événements noyau n'ont pas de mutex associé, et ne sont donc pas faits pour garder un seul *thread* actif. Enfin il n'est pas garanti que le premier *thread* en attente sera le premier réveillé.

La structure `OVERLAPPED` Tous les appels systèmes asynchrones prennent en paramètre un pointeur vers une structure `OVERLAPPED`, allouée par l'utilisateur :

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;           /* champ interne */
  ULONG_PTR InternalHigh;       /* champ interne */
```

6. Certains paramètres se choisissent à l'ouverture d'un descripteur de fichier, ou lors de certains appels systèmes qui modifient des propriétés du descripteur. Utiliser plusieurs méthodes différentes dans un même programme nécessite de ne pas mélanger les descripteurs, et maintenir le code peut être plus difficile.

```
  union {
    struct {
      DWORD Offset;
      DWORD OffsetHigh;
    };
    PVOID  Pointer;             /* champ interne */
  };
  HANDLE    hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Elle est composée de trois parties. La première partie de la structure est constituée de deux champs internes (utilisés par le système). L'un d'eux (`Internal`) contient le code d'erreur de l'opération, indiquant à tout moment sont statut : en cours, terminée avec succès, terminée avec erreur. Ces champs doivent être initialisés à 0 lors de l'appel système. La deuxième partie indique au système, si le descripteur de fichier le permet, à quel déplacement (*offset*) effectuer la lecture ou l'écriture [7]. La dernière partie est le champ `hEvent`, un pointeur vers un événement noyau. Lorsque l'opération asynchrone se termine, l'événement est signalé. Ce champ peut être `NULL`.

Ces champs ne doivent pas être modifiés pendant l'exécution de l'opération asynchrone. En conséquence, il faut utiliser une structure par opération asynchrone.

A.1 `WaitForMultipleObjects` et `select`

Les fonctions `WaitForSingleObject` et `WaitForMultipleObjects` fournies par l'API Windows permettent d'attendre un ou plusieurs événements. Elles s'utilisent sur beaucoup de types d'objets différents, notamment les descripteurs de fichier et les événements noyau.

Afin de savoir qu'une entrée/sortie asynchrone est terminée, il faut faire pointer le champ `hEvent` de la structure `OVERLAPPED` correspondant sur un événement noyau. On peut alors attendre sur cet événement.

La fonction `WaitForMultipleObjects` offre donc une interface proche de `select` sur Unix. Toutefois, elle ne permet d'attendre que sur `MAXIMUM_WAIT_OBJECTS` objets, défini à 64 dans les fichiers d'en-tête : il ne peut être utilisé en général.

L'appel système `select` est aussi présent sous Windows. Comme pour la version Unix, il prend en paramètre une liste de descripteurs de fichiers, et renvoie lesquels sont prêts pour une lecture ou écriture. Cependant, sous Windows, ces descripteurs ne peuvent être que des *sockets* synchrones (en particulier, `select` ne s'utilise pas sur des fichiers). Ce n'est pas une limitation par rapport aux Unix, puisque ces derniers retournent toujours « prêt » pour un descripteur représentant un fichier sur disque.

7. Il y a deux champs car le type `DWORD` désigne un entier non signé de 32 bits, ce qui n'est pas suffisant pour adresser de gros fichiers.

L'appel système `select` est limité à un nombre défini à la compilation de *sockets* sur lesquelles attendre. Par défaut, cette valeur est de 64, mais on peut la changer en définissant la macro `FD_SETSIZE` à la valeur désirée. La documentation Windows souligne que le fait que cette limite soit fixe est un inconvénient de `select`, et certains auteurs [8] pensent qu'il n'est pas impossible que `select` ait une implémentation liée à `WaitForMultipleObjects`, ce qui limiterait l'utilisation réelle de `select` à seulement 64 descripteurs. J'ai essayé de vérifier cette hypothèse, sans succès : dans mes tests, `select` réagissait à chacun des descripteurs en attente, et permettait donc d'attendre jusqu'à `FD_SETSIZE` descripteurs.

A.2 Les *I/O Completion Ports (IOCP)*

Un *I/O Completion Port* est une structure de données sur laquelle est basé un mécanisme de notification complexe, permettant à un programme d'une part de recevoir des messages provenant de ses *threads* et du système, et d'autre part d'avoir un nombre actif de *threads* proche du nombre de cœurs de la machine.

L'IOCP est une file de messages, appelés *paquets de complétion*. Cette file est fiable en présence de *threads (thread-safe)* : deux *threads* natifs peuvent opérer dessus simultanément sans la corrompre. Par ailleurs, l'IOCP n'est pas implémentable en espace utilisateur : certaines de ses propriétés nécessitent de pouvoir surveiller l'état des *threads*.

Afin de pouvoir être averti par un IOCP qu'une entrée/sortie asynchrone sur un descripteur de fichier s'est terminée, il faut d'abord lier le descripteur à l'IOCP. Après cela, le programme effectue son entrée/sortie asynchrone, et continue son exécution. Une fois l'opération asynchrone terminée, le système enfile un paquet de complétion dans l'IOCP. Le programme peut alors récupérer le paquet en le retirant de l'IOCP.

Dans la suite de cette partie, je présente les IOCP un peu plus en détail : d'abord, nous voyons comment créer un IOCP et le lier à un descripteur de fichier, puis comment fonctionne le mécanisme de notifications, et enfin nous parlerons de la gestion du parallélisme avec les IOCP.

Création d'un IOCP La fonction `CreateIoCompletionPort` permet de faire deux opérations : créer un IOCP, et associer un IOCP à un descripteur de fichier.

Un IOCP est un objet stocké en mémoire utilisateur, alloué par l'utilisateur. On peut ainsi créer plusieurs IOCP dans un seul processus. Au moment où on le crée, il faut lui spécifier une valeur de concurrence : nous expliquerons ce paramètre ci-dessous.

Puisqu'il peut y avoir plusieurs IOCP, le système doit, lorsqu'une opération asynchrone se termine, savoir vers quel IOCP envoyer le paquet de complétion : il faut associer chaque descripteur de fichier à un IOCP. Lors de cette association,

8. http ://www.tangentsoft.net/wskfaq/advanced.html

une clef doit être précisée. Celle-ci sera retournée en même temps qu'un paquet de complétion venant de ce descripteur.

Un mécanisme de notification Le système peut poster des paquets de complétion dans la file de l'IOCP. Ils représentent la fin d'une entrée/sortie, et sont composés de trois informations : le nombre d'octets envoyés ou reçus, la clef associée au descripteur de fichier, et un pointeur vers la structure `OVERLAPPED` relative à l'opération asynchrone.

L'utilisateur peut aussi ajouter un message à la file d'un IOCP en utilisant la fonction `PostQueuedCompletionStatus`. Le message n'est alors pas à proprement parler un paquet de complétion, aussi il n'est pas nécessaire que le pointeur pointe effectivement vers une structure `OVERLAPPED`.

Un *thread* peut récupérer un paquet de complétion d'un IOCP par un appel à la fonction `GetQueuedCompletionStatus`. Dans le cas où aucun paquet n'est disponible au moment de l'appel, cette fonction bloque pendant un temps défini en paramètre. Ce temps est défini en millisecondes ; il peut être nul ou infini.

Gestion du parallélisme Un IOCP fournit une forme de *threadpool* intégrée : il permet au système d'assigner des *threads* à l'IOCP, et d'en gérer l'exécution. La documentation Windows explique que le but recherché est d'une part d'avoir au moins autant de *threads* actifs que de cœurs processeur de la machine, afin de ne pas perdre de temps de calcul, et d'autre part d'éviter de dépasser ce nombre, afin de ne pas avoir d'inutiles changements de contextes (passage d'un *thread* à l'autre).

La valeur de concurrence optimale, spécifiée lors de la création d'un IOCP, est la valeur qui indique au système combien on désire voir de *threads* affectés à l'IOCP s'exécuter en parallèle. La valeur de concurrence (sous-entendu courante), désigne le nombre de *threads* actifs (qui ne bloquent pas) affectés à l'IOCP. Notons que la valeur de concurrence de l'IOCP sera aussi affectée par des blocages indépendants de l'IOCP. Par exemple, si un *thread* bloque en appelant la fonction `Sleep`, il sera considéré comme inactif.

Quand un *thread* demande un paquet de complétion à un IOCP, le système associe le *thread* à l'IOCP, et le met en attente dans la pile de *threads* de l'IOCP. Il est dépilé avec un paquet de complétion et reprend son exécution si et seulement si la valeur de concurrence de l'IOCP n'est pas atteinte.

A.3 Les *Completion Routines*

Les *Completion Routines* sont des fonctions passées en paramètre à certains appels systèmes afin qu'elles soient exécutées après réalisation de l'opération. C'est un mécanisme indépendant des IOCP, et qui est incompatible avec ce dernier : on ne peut pas utiliser les *Completion Routines* avec un descripteur de fichier lié à un IOCP.

Les *completion routines* sont un mécanisme de *callback* pour les entrées/sorties asynchrones : lorsque l'opération asynchrone sera terminée, le système exécutera

la *completion routine* passée en paramètre de l'appel système. Les *completion routines* sont implémentées avec les APC. Nous voyons dans la suite de cette partie le mécanisme des APC, ses spécificités, et les conséquences sur l'exécution des *completion routines*.

Asynchronous Procedure Call (APC) Une APC est une fonction qui sera exécutée par un *thread* choisit par l'appelant. Il en existe deux types : les APC noyau (*kernel-mode APC*) et utilisateur (*user-mode APC*). La différence entre les deux se situe sur l'endroit dans le code où elles pourront être exécutées : une APC utilisateur ne pourra être exécutée que lorsque le *thread* détenteur sera alerte et en attente *(waiting and alertable state)*[9], alors qu'une APC noyau pourra aussi être exécutée lorsque le *thread* reprend son exécution (après avoir été interrompu par l'ordonnanceur). Un programme peut ajouter une APC à un *thread* par la fonction `QueueUserAPC`, en lui donnant en paramètre le descripteur de fichier correspondant au *thread*.

Bien que ce comportement ne soit pas documenté, j'ai observé que lorsqu'un *thread* commence à exécuter ses APC, il les exécutera jusqu'à ce qu'il n'y en ait plus, y compris celles qui seraient ajoutées pendant l'exécution, et cela indépendamment du point de coopération. Par exemple, si on désire faire une attente d'une seconde, mais exécuter des APC dans cet intervalle de temps, on utilisera naturellement la fonction `SleepEx`, mais l'exécution des APC ne sera pas interrompue au bout d'une seconde. Inversement, si l'exécution dure moins longtemps que l'attente prévue, la fonction retourne quand même, avec un message d'erreur indiquant l'interruption.

Par ailleurs, l'appel à une APC se fait au dessus de la pile courante du *thread*, et lors de son exécution, elle opère comme si elle était le *thread* lui même : on peut notamment se mettre dans un état d'attente alerte, en quel cas la pile d'APC est exécutée à ce moment là. Il faut noter qu'une APC prend sur la pile une place considérable (de l'ordre de 1000 octets), ce qui n'est pas gênant, car utiliser récursivement des APC n'est pas une option, la file des APC du *thread* devant être épuisée avant de retourner.

Limites Le mécanisme des APC manque de souplesse : le programme ne peut pas choisir d'exécuter un nombre fini d'APC, ni pendant un temps donné. La réactivité du programme peut en être affecté.

Dans un environnement à plusieurs *threads*, garantir l'équité est difficile : un *thread* qui aurait terminé d'exécuter sa pile d'APC ne peut pas décharger un autre *thread*, en partageant son travail.

Dans mes recherches, j'ai trouvé des morceaux de code qui cherchaient à rendre les APC plus réactives, en allant modifier des données internes du *thread* pour que le système croit qu'il soit toujours dans l'état d'attente alerte. Cette modification permettait à la routine d'être appelée immédiatement, et de ne pas avoir à attendre

9. Seul l'appel à certaines fonctions permettent d'être en attente alerte.

que le *thread* soit dans un état particulier [10].

Completion routines Les *completion routines* sont des fonctions particulières passées lors d'un appel système asynchrone afin qu'elles soient exécutées à la fin de l'opération.

Le système appelle ces fonctions en utilisant les APC, et choisit pour *thread* celui qui a émis l'appel système. Utiliser les *completion routines* hérite donc des propriétés des APC, et ajoute un inconvénient : le programmeur n'a pas la possibilité de répartir la charge sur ses différents *threads*, ce qui accentue les problèmes d'équité.

Enfin, le type des *completion routines* dépend des appels systèmes : par exemple, celle passée à `WSARecv` prend un argument de plus que celle passée à `ReadFileEx`.

A.4 Autres méthodes

Attente sur un descripteur de fichier Un programme peut se mettre en attente sur un descripteur de fichier avec la fonction `WaitForMultipleObjects`. Si une opération asynchrone est en cours sur ce descripteur, la fonction retournera lorsque l'opération asynchrone sera terminée. Cette méthode ne permet pas de distinguer quelle opération asynchrone termine.

Les *ThreadPools* L'utilisation des *threadpools* semble être une combinaison ou une alternative aux autres implémentations. Je n'ai pas eu le temps d'étudier en profondeur les mécanismes à base de *threadpools*, ni de les tester. Il y a d'ailleurs plusieurs façons de faire des entrées/sorties avec les *threadpools*.

Il semble qu'elles soient implémentées en utilisant un IOCP, d'une part, et bénéficient d'une méthode particulière pour la gestion des APC.

Comme pour les IOCP, il est possible de lier une *threadpool* à un descripteur de fichier, mais en lui donnant la fonction à exécuter pour tous les paquets de complétion venant de ce descripteur.

Le véritable intérêt des *threadpools* semble être pour les *completion routines* : une *threadpool* répartirait les APC sur les différents *threads*, ce qui résoudrait les problèmes d'équité décrits ci-dessus.

Autre Il existe encore beaucoup de fonctions qui permettent plus ou moins les mêmes choses que les mécanismes déjà décrits. Par ailleurs, ceux-ci peuvent parfois se combiner, avec des résultats variables.

10. http ://www.codeproject.com/Articles/7238/QueueUserAPCEx-Version-2-Truly-Asynchronous-User-M

B Description des serveurs

Completion routines J'ai développé deux versions utilisant les *completion routines* : une qui utilise un `accept` synchrone, et une autre un `accept` asynchrone.

`accept` synchrone La boucle à événements de ce serveur est centrée sur l'appel système `accept`. Celui-ci accepte autant de connexions que possible (jusqu'à ce que la file noyau d'`accept` soit vide). Lorsqu'une connexion est reçue, le serveur commence à la traiter, jusqu'à la première entrée/sortie. Celle-ci est asynchrone, et la *completion routine* correspondant à la continuation est passée à l'appel système servant à faire l'entrée/sortie.

Lorsque la file d'`accept` est vide, l'appel système `accept` bloque, et met le *thread* courant en attente alerte, c'est-à-dire, rappelons-le, que la file d'APC est exécutée.

`accept` asynchrone Seule la fonction `AcceptEx` permet d'accepter les connexions de manière asynchrone, et elle ne permet pas l'utilisation des *completion routines*. J'ai donc choisi d'utiliser les IOCP pour accepter les connexions de ce serveur, et notamment la fonction `GetQueuedCompletionStatusEx`, qui permet au *thread* l'attente alerte de paquets de complétion. Cette fonction constitue le cœur de la boucle principale.

Comme nous l'avons dit, les mécanismes des *completion routines* et les IOCP sont incompatibles (A.3). Aussi, seule la *socket* sur laquelle on fait des `AcceptEx` est liée à l'IOCP. Par ailleurs, il n'est nul besoin d'utiliser les *completion routines* sur des *sockets* correspondant aux clients.

Un certain nombre d'appels asynchrones à `AcceptEx` sont initiés dès le lancement du serveur, après quoi on entre dans la boucle principale. Lorsqu'une connexion se termine, un autre appel à `AcceptEx` est lancé, maintenant le nombre de connexions maximum à une constante prédéfinie.

L'appel système `AcceptEx` permet aussi de recevoir les premières données envoyées par le pair : `AcceptEx` combine un `accept` et un `recv` de manière asynchrone. Par ailleurs, la fonction `AcceptEx` doit recevoir une *socket* déjà créée.

IOCP J'ai écrit trois serveurs exploitant les IOCP. Dans tous les cas, la boucle à événements est centrée sur l'appel à `GetQueuedCompletionStatus`. Les serveurs se distinguent principalement sur leur manière d'accepter les connexions entrantes.

`accept` synchrone La fonction `accept` est nécessairement synchrone, mais particulièrement simple à utiliser. Afin de pouvoir l'utiliser sans bloquer la boucle à événements, j'ai dédié un *thread* à cet usage.

Dans cette version, un *thread* ne fait que des `accept`, et un autre exécute la boucle à événements. La *socket* nouvellement créée par l'appel système `accept` est

transmise à la boucle à événements en utilisant `PostQueuedCompletionStatus` (voir paragraphe A.2 page 23).

`accept` asynchrone Comme pour les *completion routines* avec accept asynchrone, on utilise pour ce serveur l'appel système `AcceptEx`.

Les deux variantes des serveurs utilisant `AcceptEx` diffèrent par leur gestion des environnements correspondant au suivi de chaque requête. L'un d'eux alloue ces structures en fonction de la demande, et l'autre prévoit statiquement le nombre de connexions que le serveur peut supporter.

`select` Le serveur à base de `select` est très similaire à celui utilisé pour les précédents serveurs : il s'agit aussi d'une boucle à événements, mais où le cœur de la boucle est constitué de l'appel système `select`. La différence principale se situe dans le fait que tous les appels systèmes effectués sont synchrones non-bloquants, et non asynchrones.

Threads Les serveurs à base de *threads* utilisent un *thread* par connexion. Chaque *thread* ayant sa propre pile, il n'y a pas d'allocation d'environnement. Afin d'obtenir des performances meilleures, la pile de chaque *thread* est précisée au moment de créer le *thread*, de telle sorte qu'elle soit relativement proche de ce que le *thread* pourrait utiliser comme mémoire.

Encore une fois, j'ai écrit deux variantes : la première dédie un *thread* pour faire des `accept`, et crée un nouveau *thread* pour chaque *socket* acceptée. Le deuxième lance au démarrage autant de *threads* que de connexions que le serveur doit pouvoir supporter ; chaque *thread* fait un `accept`, puis dessert le client accepté, et enfin boucle.

La documentation Windows indique que la taille de la pile d'un *thread* est typiquement arrondie au multiple de 64Ko supérieur : même si on demande une petite pile, elle sera relativement importante. Dans le cas où tous les *threads* sont lancés au démarrage, on observe assez vite une limite sur le nombre de *threads* que le système peut allouer. Sur la machine où les courbes sont produites, `CreateThread` échoue à environ 1000 *threads*.

C Résultats de tous les serveurs

Dans cette section, nous présentons les résultats détaillés de tous les serveurs, avec deux résultats à chaque fois, qui, bien que sous les mêmes conditions, donnent des résultats différents. Les conditions expérimentales et les représentations sont les mêmes que dans le reste du document (partie 1.5 page 8). Les graphiques présentés ici incluent les courbes de la partie 1.5.2. Ils sont aussi réalisés avec 50 000 requêtes par test.

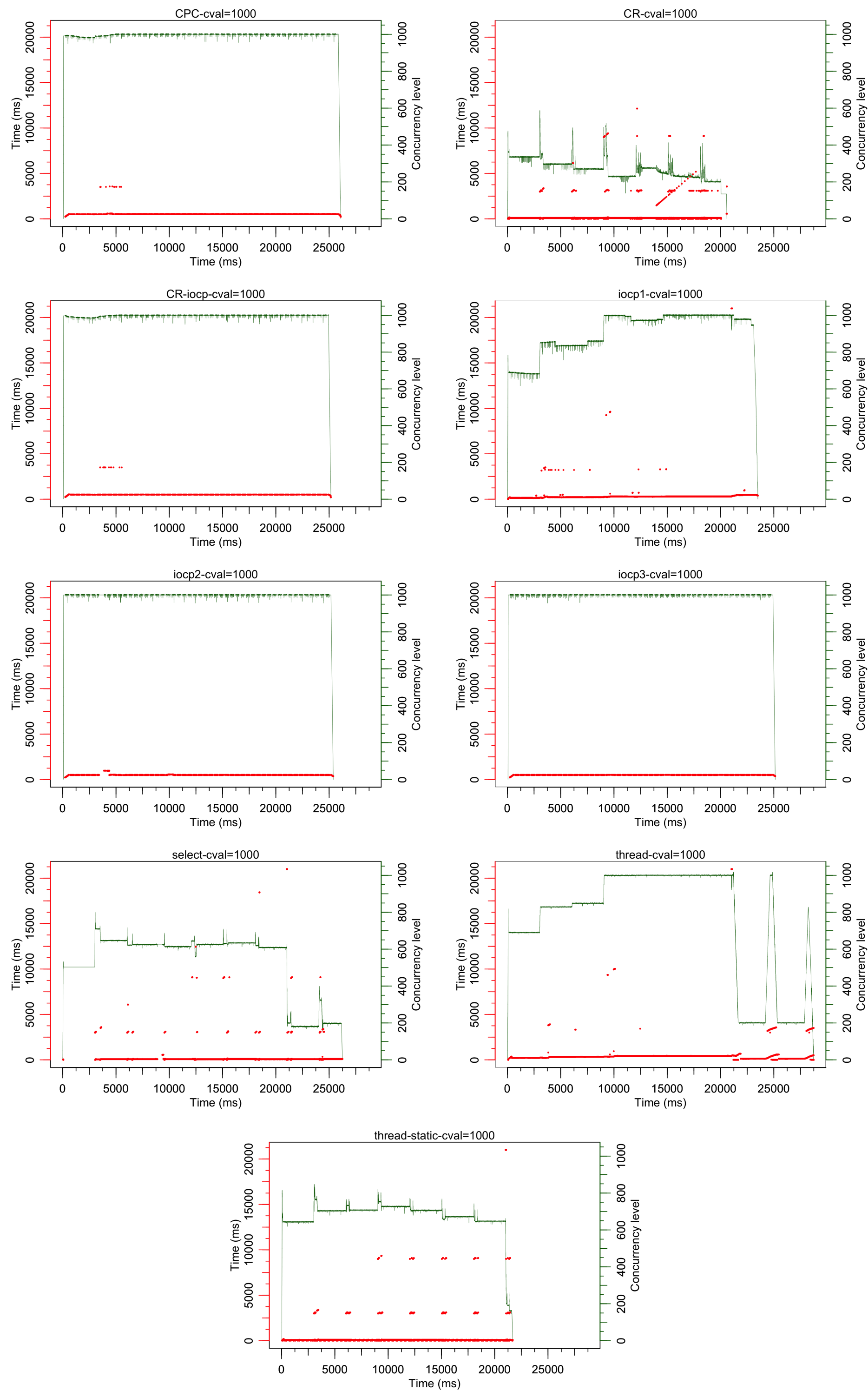

FIGURE 4 – Jeu de tests de serveurs Web

Conditions expérimentales Les tests ont été réalisés avec un serveur Intel Pentium M à 1,70 Ghz, 1Go de mémoire vive, et une interface *Ethernet* à 100 Mb/s. Il s'exécute sous Windows 7 Professionnel. Le client est un AMD Athlon 64 3500+ à 2,20 GHz, 1Go de mémoire RAM. Il s'exécute sous Debian, version 6.0.3. Le client teste le serveur avec apacheBench, un logiciel fourni avec le serveur web Apache permettant de tester les performances d'un serveur web. Toutes les mesures réalisées se font côté client.

Représentations Sur chaque graphique, deux données sont représentées, avec en abscisse le temps depuis le début de l'expérience. La première est un nuage de points. Chaque point représente l'accomplissement d'une requête. En ordonnée (échelle de gauche) se trouve le temps qu'a mis la requête avant d'être complétée. La deuxième est le niveau de concurrence côté serveur en fonction du temps, c'est-à-dire le nombre de connexions que le serveur traite simultanément.

Interprétation La figure 4 représente les résultats détaillés d'un jeu de test, représentatif des comportements observés.

On remarque tout d'abord (courbe du dessus) que les seuls serveurs à assumer la charge tout au cours du temps sont ceux utilisant les `accept` asynchrones (CPC, *completion routines* avec IOCP, IOCP2 et IOCP3).

Le nuage de points nous montre comment le serveur se comporte pour chaque requête. On voit pour presque tous les serveurs (regardons par exemple le premier : CPC) qu'à environ trois secondes, quelques requêtes terminent, et ont mis environ trois secondes à s'accomplir. Ces requêtes ont donc été émises au début du test. Elles ont été réémises, car on peut observer que l'acquittement de demande de connexion se fait au à environ trois secondes [11] : ces requêtes n'ont pas été traitées plus lentement que les autres par le serveur.

Tous les serveurs ont ce problème de paquets perdus lors d'un trop grand nombre de requêtes simultanées. Cependant, il arrive que les serveurs avec `accept` asynchrones arrivent à ne pas en perdre. C'est ici le cas des serveurs IOCP2 et IOCP3.

Les serveurs utilisant les `accept` asynchrones ne perdent plus de paquets après la rafale initiale. Les autres serveurs en revanche en perdent régulièrement. En particulier, on voir qu'avec le serveur utilisant `select`, certains paquets sont même réémis plusieurs fois.

La figure 1 (section 1.5.1 page 10) compare tous les serveurs en terme de temps moyen par requêtes. Comme expliqué dans cette même partie, les serveurs n'utilisant pas les `accept` asynchrones produisent des erreurs pour certaines requêtes, ce qui explique la perte de progression linéaire de ces serveurs. Cependant, cette figure permet de voir que les serveurs avec `accept` asynchrones se comportent parfaitement, et qu'ils sont tous trois très similaires en termes de performances. En particulier, le serveur à base de CPC est à peine plus lent que les meilleurs serveurs.

11. Le délai de réémission d'un paquet TCP sans acquittement est de trois secondes.

Tests sur gros fichiers Les tests précédents étaient réalisés sur un petit fichier, qui correspond à une petite page html. J'ai aussi effectué des tests avec un gros fichier (700Ko), mais seulement vers la fin de mon stage.

Le comportement des différents serveurs est très différent d'avec un petit fichier. Tous les serveurs se ressemble fortement. En effet, ils ont tous le temps de se stabiliser pour atteindre le niveau de concurrence demandé. On constate, lorsque beaucoup de requêtes se terminent en même temps, que le nombre de connexions simultanées descend fortement, et pendant un temps relativement long, avant de remonter. Curieusement, ce phénomène est plus important avec les serveurs utilisant des `accept` asynchrones.

D'un point de vue de l'efficacité, le temps moyen d'exécution des requêtes oscille entre 59 et 60 secondes en fonction des serveurs, les laissant donc presque à égalité. Tous traitent quelques requêtes beaucoup plus rapidement que la majorité des autres, et on observe aussi parfois des requêtes plus lentes. Cependant, je n'arrive pas à distinguer de comportement particulier entre les serveurs.

Enfin, les erreurs notifiées par le client (apacheBench) apparaissent pour tous les serveurs, y compris ceux avec `accept` asynchrones. Toutefois, il semble que les erreurs soient plus régulières pour ceux-ci, et relativement peu élevées. Pour les autres serveurs, il semble que cela dépende des tests. Voici le nombre d'erreurs relevées par le client sur trois tests, à un niveau de concurrence de 1000, pour 10 000 requêtes. Il y a plus de 10 000 erreurs pour un des serveurs car apacheBench somme trois types d'erreurs. Je suppose qu'une même requête peut donner lieu à plusieurs types d'erreurs.

CR-iocp	138	87	276
CR	30	602	846
iocp1	659	662	138
iocp2	15	114	27
iocp3	48	99	42
select	675	2052	812
thread	10553	1242	1330
thread-static	81	315	351
CPC	81	111	90

Ces résultats surprenants nécessiteraient des tests supplémentaires afin de mieux appréhender le comportement des différents serveurs, et plus de temps pour pouvoir mieux les comprendre.

D Comparatifs : IOCP contre file utilisateur

Je m'intéresse dans cette partie à comparer les performances des IOCP en tant que file. Pour cela, j'ai implémenté un même programme qui dispose de plusieurs variantes d'opérations d'ajout et de retrait d'éléments dans des files. J'en ai fait quatre versions différentes : une première avec les IOCP, utilisant la fonction

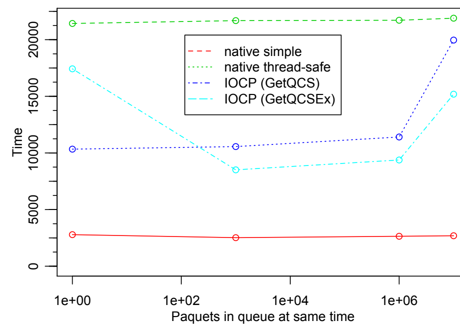

FIGURE 5 – Comparatifs : IOCP et files utilisateurs

`GetQueuedCompletionStatus`, qui permet de retirer un paquet de complétion, la deuxième avec les IOCP, et la fonction `GetQueuedCompletionStatusEx`, qui permet de retirer plusieurs paquets en un seul appel, la troisième étant une simple file utilisateur, et la quatrième étant une file utilisateur utilisant des mutex. Toutes ces implémentations véhiculent les mêmes données : mes files utilisateurs prennent donc plusieurs champs en paramètres (correspondant à ceux véhiculés par les IOCP), et les copient dans la file.

Je m'intéresse aussi au comportement des différentes files en fonction de leur saturation. Pour cela, le programme prend en paramètre une valeur de concurrence. Il commence par ajouter des paquets jusqu'à obtenir autant de paquets dans la file que cette valeur, puis ajoute et retire successivement un paquet, jusqu'à ce qu'on ait fait le nombre d'opérations voulues.

La figure 5 représente le temps d'exécution du programme, qui ne fait qu'ajouter et retirer 10 000 000 éléments d'une des files. L'échelle en abscisse est logarithmique. L'abscisse représente le nombre de paquets présents dans la file.

La figure montre que la fonction `GetQueuedCompletionStatus` est plus rapide que `GetQueuedCompletionStatusEx` pour retirer peu de paquets de la file des IOCP, tendance qui s'inverse rapidement avec plus de paquets, mais pour un gain relativement faible. On remarque aussi que lorsqu'énormément de paquets sont dans la file des IOCP (de l'ordre de quelques millions), la file des IOCP devient beaucoup plus lente, quelle que soit la fonction utilisée.

Les IOCP sont près de quatre fois plus lents qu'une file utilisateur simple sans mutex. Cependant, ils sont tout de même deux fois plus rapides qu'une file utilisateur avec mutex. On remarque aussi que les files utilisateurs ne sont pas ralenties lorsqu'elles sont surchargées (au moins pour ces niveaux de surcharge).

E Autres tests

E.1 *Sockets* dans l'état *Time Wait* et SO_REUSEADDR

Dans une connexion TCP, le pair qui a décide la fermeture retient pour un certain temps l'adresse IP et le port de connexion de la *socket* liée à l'autre pair, afin d'empêcher toute autre *socket* de se lier à cette même adresse et de ce même port. On dit que la *socket* qui vient d'être fermée est dans l'état *Time Wait*. Toutefois, on peut permettre à une *socket* de se lier à l'adresse et au numéro de port associés à une *socket* dans l'état *Time Wait*. Pour cela, il faut appeler la fonction `setsockopt` avec l'argument `SO_REUSEADDR` sur la *socket* à lier.

Dans le cas d'un serveur web, c'est le serveur qui ferme la connexion, lorsqu'il a fini d'envoyer le fichier [FTY99]. Etant donné que nous réalisons nos tests avec un seul client, l'adresse IP ne change pas, et très vite, tous les numéros de ports sont atteints. La commande `netstat` permet de voir les *sockets* dans l'état *Time Wait* : j'ai constaté que beaucoup de *sockets* sont cet état après le lancement d'un test.

Utilisation d'un délai Le système dispose d'environ 20 000 numéros de ports, et nos tests s'effectuaient chacun sur 10 000 requêtes. Afin d'essayer de corriger des problèmes, j'ai essayé de mettre un délai entre deux tests. Le délai d'expiration de l'état *Time Wait* est de 120s. Les résultats étaient bien meilleurs, mais pourtant le problème n'était pas corrigé pour tous les tests.

Influence de `SO_REUSEADDR` J'ai testé l'influence de `SO_REUSEADDR` sur un serveur à base d'IOCP avec `accept` asynchrone, en utilisant quatre variantes : sans utiliser `SO_REUSEADDR`, en l'utilisant avant d'utiliser `AccetpEx` sur la *socket*, en l'utilisant après avoir accepté la connexion, et en l'utilisant dans les deux cas. Dans chaque test, je demande 50 000 requêtes avec un niveau de concurrence de 1000. Le système dispose d'environ 20 000 numéros de ports, le délai d'expiration de l'état *Time Wait* est de 120s, et la plupart des tests sont terminés au bout de 25s, aucun n'excède 35s.

Dans les quatre cas, les résultats sont sensiblement les mêmes, et présentent les mêmes symptômes. Par ailleurs, bien que le système retienne effectivement beaucoup de *sockets* dans l'état *Time Wait*, aucun blocage susceptible d'être en relation avec ce phénomène n'est observé.

Le fait qu'aucun blocage ne se soit ressenti dans ces tests laisse à penser que l'obtention de meilleurs résultats par l'utilisation d'un délai était une coïncidence. En effet, avec 50 000 requêtes en quelques secondes, tous les numéros de ports devraient être occupés.

E.2 Optimisations d'appels systèmes pour les entrées/sorties

La documentation Windows (*msdn*) nous informe qu'il est plus rapide, pour lire sur une socket, d'utiliser `WSARecv` que `ReadFile`. D'autre part, on sait qu'un appel

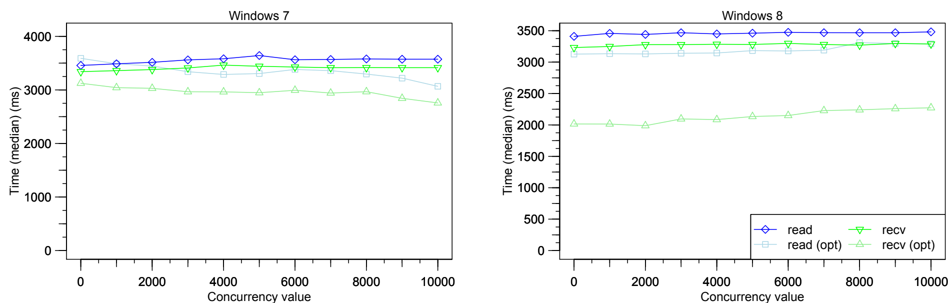

Figure 6 – Comparaison des performances en fonction de la charge

système asynchrone peut se comporter de manière synchrone lorsque les données sont prêtes, et on peut paramétrer un descripteur de fichier pour que dans le cas synchrone, aucun paquet de complétion ne soit posté. La prise en compte de ce cas améliorerait aussi les performances, selon la même documentation. Cependant, je n'ai pas trouvé de comparatifs : il m'importait de le vérifier, et le quantifier.

Programme testé J'ai écrit un programme réalisant des lectures et écritures sur une *socket* locale au programme, avec deux variantes : l'une utilisant `ReadFile` et `WriteFile`, l'autre utilisant `WSARecv` et `WSASend`. Le programme a deux paramètres : le nombre d'entrées/sorties désirées, et un niveau de concurrence. Le programme commence par initier un certain nombre de lectures et d'écritures au début du programme (le niveau de concurrence), puis, lorsqu'une notification venant d'une lecture ou d'une écriture est reçue, ajoute respectivement une lecture ou une écriture, jusqu'à ce que le nombre d'entrées/sorties désirées soit terminé.

Conditions expérimentales Le programme a été testé sur Windows 8 Consumer preview, en machine virtuelle, montée sur un MacBook Pro 2, 53Ghz Intel Core 2 Duo à 4Go de mémoire, utilisant 1 cœur et 1,5Go de mémoire. Il a aussi été testé, comme pour les serveurs, avec Windows 7, installé sur un DELL Inspiron 8600 avec un processeur Intel Pentium M à 1, 70 Ghz, 1Go de mémoire vive. Dans les deux cas, on mesure le temps d'exécution du programme, avec la commande `Measure-Command` du *PowerShell* de Windows.

Représentation et interprétation Les résultats de la figure 6 confirment qu'utiliser les fonctions `WSARecv` et `WSASend` est respectivement plus rapide que `ReadFile` et `WriteFile`, et que considérer le cas synchrone permet encore d'améliorer la rapidité. Les deux optimisations combinées permettraient d'être environ 14% plus rapide sous Windows 7, et près de 45% plus rapide pour Windows 8. Cette dernière différence est étonnante, car chaque amélioration prise à part ne permet de gagner

que quelques pourcents, mais c'est la combinaison des deux qui permet d'atteindre ces performances.

E.3 L'appel système `select` et `FD_SETSIZE`

Certains auteurs affirment que `select` pourrait être implémenté en utilisant des fonctions proches de `WaitForMultipleObjects`, limitant sa capacité d'écoute à 64 descripteurs.

Afin de voir si `select` restait réactif quel que soit le nombre de descripteurs de fichiers, je me suis connecté au server utilisant `select` que j'ai écrit, par *telnet*, avec beaucoup de terminaux. J'ai ensuite sélectionné un terminal au hasard, et envoyé des données au serveur. Celui-ci m'a toujours répondu : je n'ai donc pas pu mettre de limites en évidence.