Réalisation d'un *seeder* BitTorrent en CPC

Stage de TRE

sous la direction de

J. Chroboczek et G. Kerneis

Pejman Attar Yoann Canal

30 juin 2009

Table des matières

1 Introduction

CPC est un langage de programmation, conçu par J. Chroboczek et prouvé correct et étendu par G. Kerneis, qui permet d'écrire des programmes concurrents. Avant notre projet, les seuls programmes écrits en CPC étaient une suite de programmes tests ainsi qu'un serveur HTTP minimaliste.

Le but de ce projet est de se lancer dans un vrai projet en CPC afin de chercher des techniques de programmations permettant de tirer partie des spécificités de ce langage.

1.1 CPC

Lors de développement logiciel, on a couramment besoin de faire réaliser au programme écrit plusieurs actions simultanément. Un exemple commun est celui des serveurs Web qui doivent servir plusieurs clients en même temps. Cette simultanéité peut être réelle, comme dans le cas de multiprocesseurs ou de calcul distribué sur plusieurs ordinateurs, ou apparente en entrelaçant les diverses tâches en profitant des pauses dans l'une pour exécuter les autres.

Un processus léger (en anglais, *thread*), est similaire à un processus dans le sens où tous deux représentent l'exécution d'un ensemble d'instructions d'un processeur. Du point de vue de l'utilisateur, l'exécution de *threads* distincts semble se dérouler en parallèle. Toutefois, là ou chaque processus possède son propre espace d'adressage (tas, ou *heap*), les processus légers d'un même processus se partagent un unique espace d'adressage. Dans les deux cas, chaque processus et chaque *thread* possède sa propre pile (*stack*).

On distingue deux façons d'ordonnancer les *threads* :

- les *threads* préemptifs, qui peuvent être interrompus à n'importe quel moment par l'ordonnanceur ;
- les *threads* coopératifs, qui passent la main quand ils arrivent à un point de coopération explicité par le programmeur.

Ces deux techniques d'ordonnancement ont des caractéristiques auxquelles il faut prêter attention :

- on ne peut pas prévoir comment les instructions des *threads* préemptifs seront entrelacées. Il faut donc utiliser des verrous, et donc des algorithmes complexes, pour assurer que deux *threads* n'accèdent pas aux mêmes données en même temps (exclusion mutuelle). Par exemple, le pseudo code suivant :

  ```
  n = read(m) ;
  write(m, n+1) ;
  ```

  exécuté simultanément par 2 *threads* donne un résultat non déterministe.
- dans le cas des *threads* coopératifs l'entrelacement des instructions est contrôlée par le programmeur qui choisit les points de coopération. Les problèmes d'exclusion mutuelle sont donc simplifiés, mais il faut faire

attention à ce que les *threads* rendent la main régulièrement (absence de famine).

Une autre approche pour la concurrence est la programmation événementielle, ou programmation par boucle à événements (*event loop*). Un programme écrit dans ce style attend que des événements extérieurs se produisent (par exemple la connection d'un client si le programme est un serveur). Lorsque un événement est reçu, la boucle invoque le gestionnaire d'événements (*handler*) qui lui est associé (par exemple la fonction qui gère la communication avec un client).

Chaque fonction doit être relativement courte pour rendre rapidement la main à la boucle pour donner l'impression de simultanéité à l'exécution de la gestion des événements (par exemple donner l'impression que les clients sont servis simultanément et non séquentiellement). Si une fonction est trop longue, on la divise en sous-tâches qui se passent la main successivement par l'intermédiaire de la boucle à événements.

La programmation par *threads* permet de visualiser assez aisément le déroulement de l'exécution car le flot de contrôle du programme est explicite. Cependant,

- un *thread* possède sa propre pile et donc prend de la place en mémoire ;
- la création d'un *thread* est une opération potentiellement coûteuse (création d'une nouvelle pile ainsi que des structures de données nécessaires à son exécution) ;
- le passage d'un *thread* à l'autre nécessite la sauvegarde et la restauration des registres du processeur (changement de contexte), ce qui est une opération potentiellement longue ;
- la destruction d'un *thread* est également potentiellement coûteuse (libération de la pile et d'autres structures de données).

En revanche, la programmation événementielle n'a pas ces inconvénients — les gestionnaires d'événements ne nécessitant pas de changement de contexte et étant rapides à créer — mais est beaucoup plus complexe à appréhender, car le flot de contrôle est scindé entre les différents gestionnaires d'événements.

CPC(Continuation Passing for C) CPC est une extension au langage C pour la concurrence conçue par J. Chroboczek et prouvée correcte par G. Kerneis. Cette extension permet, par l'ajout d'un certain nombre de primitives, de manipuler des *threads*. CPC permet des programmes dans un style de programmation par *threads* coopératifs qui seront transformés à la compilation en une boucle à événements. CPC permet donc d'allier l'efficacité de la programmation par boucle à événements et la clarté relative de la programmation par *threads*.

À la différence des *threads* natifs, les *threads* cpc :

- sont extrêmement légers (20 octets contre quelques dizaines de kilo-

octets) ;
- sont extrêmement rapides à créer (environ 100 fois plus rapide que ceux de la librairie NPTL (librairie de *threads* POSIX sous Linux) ;
- sont coopératifs (les *threads* natifs sont préemptifs) ;
- permettent des changements de contexte extrêmement rapides.

Comme les *threads* cpc sont légers et ne coûtent rien a créer, ils sont particulièrement adaptés pour les programmes qui manipulent un nombre massif de *threads* (comme les serveur Web).

Une autre différence sémantique entre les *threads* natifs et les *threads* cpc est que ces derniers n'ont pas d'identifiant comme c'est le cas dans les librairies pthread ou ST. On ne donc peut pas leur envoyer des signaux Unix. La librairie standard de cpc ne possède donc pas de fonctions comme `pthread_kill`.

Cependant, avoir la possibilité de manipuler des *threads* natifs est essentiel, par exemple pour exécuter des appels à des fonctions bloquantes comme `gethostbyname`. Dans ce cas, utiliser des *threads* cpc bloquerait l'ensemble de notre programme. Pour cette raison, CPC permet la manipulation de *threads* natifs.

Du point de vu utilisateur, on ne créé pas de *thread* natif mais on peut rendre un *thread* cpc natif grâce à la primitive `cpc_detach`. À l'inverse, pour rendre un *thread* détaché coopératif, on utilise la primitive `cpc_attach`.

La création et destruction successive de nombreux *threads* natifs aurait un effet désastreux sur les performances. Pour palier à ce problème, CPC maintient un ensemble de *threads* natifs (ou *thread pool*). Cet ensemble est vide tant qu'aucun appel à `cpc_detach` n'a été fait. Chaque fois qu'un *thread* cpc fait un appel à `cpc_detach`, si la *thread pool* contient un *thread* natif libre on le réutilise, sinon on en créé un. Lorsque le *thread* fait appel à `cpc_attach`, le *thread* natif est libéré mais n'est pas détruit avant un certain temps (une seconde dans l'implémentation actuelle de CPC). Cette technique permet d'utiliser répétivement `cpc_detach`/`cpc_attach` en ne payant le coût de la création et de la destruction d'un *thread* natif qu'une seule fois. De plus, la *thread pool* ne peut contenir qu'un nombre fini de *threads* natifs au-delà duquel les *threads* faisant appel à `cpc_detach` sont mis en attente. Ce mécanisme permet de limiter le nombre de *threads* natifs simultanés et donc de limiter le coût lié à de nombreux changements de contexte.

Variables de condition Dans CPC, les variables de condition sont la primitive principale de synchronisation des *threads*. Les variables de condition sont un moyen de faire attendre un changement d'état à un ensemble de *threads*. Lorsqu'un *thread* a besoin d'attendre que le programme soit dans un certain état pour continuer son exécution, il se met en attente sur la variable de condition correspondant à cet état et entre dans une liste de *threads* en attente sur cette variable de condition.

Lorsqu'un autre *thread* remarque que les conditions voulues sont réunies, il peut signaler la variable de condition de deux façons différentes :

- soit il réveille le premier *thread* de la liste de *threads* en attente sur cette variable de condition (`cpc_signal`),
- soit il réveille l'ensemble des *threads* en attente (`cpc_signal_all`).

Barrières Les barrières sont une autre technique de synchronisation des *threads* CPC. Elles ne sont pas des primitives du langages mais implémentées dans la librairie standard à l'aide de variables de conditions.

Une barrière est une variable sur laquelle peuvent attendre les *threads*. La barrière est crée avec un nombre de *threads* à synchroniser en argument. Les *threads* sont tous replacés dans la file d'exécution dès lors que le nombre prédéfini de *threads* en attente sur la barrière est atteint.

limitations de l'implémentation Le langage et le compilateur CPC étant encore au stade expérimental, nous avons rencontré quelques problèmes mais, comme nous étions encadrés par leurs concepteurs, ces problèmes ont étés résolus à chaque fois dans un délai très court.[1]

1.2 BitTorrent

BitTorrent est un protocole de transfert de fichiers pair à pair. Il ne se préoccupe pas de la recherche de contenu mais uniquement de sa diffusion. À la différence de protocoles de téléchargement client/serveur (comme FTP ou HTTP) la ressource n'est pas centralisée sur une seule machine mais répartie sur un ensemble de pairs (l'essaim, ou *swarm*). On peut voir chaque pair comme un serveur partiel, c'est à dire qu'il peut ne servir que la fraction de la ressource qu'il possède. De cette façon la disponibilité de la ressource augmente avec la demande au lieu d'être divisée proportionnellement à la demande comme dans le cas d'un protocole client/serveur.

L'*overhead* dû au protocole BitTorrent est négligeable (de l'ordre de l'octet par kilo-octet de donnée) et donc, dans le pire des cas — celui d'un torrent peu populaire ou qui vient d'être rendu disponible — un *seeder* BitTorrent est aussi efficace qu'un serveur utilisant un protocole client/serveur.

Pour fonctionner, BitTorrent a besoin de deux entités autres que l'essaim : un *tracker* et un fichier de méta-données.

Le *tracker* est un serveur ne faisant pas partie de l'essaim et qui n'envoie pas de données. Son rôle est d'annoncer à la demande des adresses de pairs participant à l'essaim pour un torrent donné.

Le fichier de méta-données, portant l'extension ".torrent", doit être téléchargé par un moyen extérieur au protocole (en général sur un serveur HTTP). Ce fichier contient toutes les informations relatives à la recherche d'autres pairs

[1] L'utilisation de gdb sur un programme CPC est intéressante.

(adresse du *tracker*), ainsi que celles relatives à l'identification du torrent et à la vérification de son intégrité. Cette intégrité est vérifiée en découpant le torrent en fragments — de spécifiée par le fichier de méta-données — et en comparant leur *hash* avec celui annoncé dans celui ci.

BitTorrent a reçu beaucoup de mauvaise presse du au fait qu'il est souvent utilisé pour diffuser du contenu illégal, mais BitTorrent est avant tout un protocole de transfert, indépendant de la nature du contenu transféré.

1.2.1 *Seeder* BitTorrent

Un *seeder* BitTorrent est comme un pair BitTorrent dont le seul but est d'envoyer des informations aux autres pairs contrairement à la plupart des client BitTorrent qui se préoccupent surtout d'optimiser la réception et non l'émission de données.

Les trois particularités de BitTorrent exposées précédemment — taille négligeable de l'*overhead*, productivité des pairs, et possibilité de servir un nombre important de torrents à un grand nombre de pairs simultanément font qu'un *seeder* est particulièrement intéressant pour les fournisseurs de contenu comme Jamendo[2], qui distribue de la musique libre, ou Blizzard[3], éditeur de jeux vidéos qui distribue ses mises à jour par BitTorrent.

À l'heure actuelle, il n'existe qu'un seul *seeder* libre (Permaseed[4]), qui n'est plus maintenu depuis 2006 et nécessite l'emploi d'un *tracker* spécifique (Osprey Web application). Nous supposons par ailleurs que certains distributeurs de contenu — entre autres Blizzard et Amazon — utilisent leurs propres *seeders* propriétaires.

2 Efficacité et passage à l'échelle

Un *seeder* n'étant guère plus qu'un pair BitTorrent tronqué — il n'est pas conçu pour recevoir des données — il doit donc faire ce pour quoi il est spécialisé beaucoup plus efficacement qu'un pair classique.

2.1 Montée en charge par rapport au nombre de clients

Un *seeder* doit être capable de supporter un très grand nombre de clients. Par exemple, le jour de la sortie de ses mises à jour, Blizzard doit supporter la connection de plus de 10 millions de joueurs le temps que l'essaim ait suffisamment de pairs productifs pour se passer de ses *seeders*.

Si on veut pouvoir supporter plusieurs milliers de clients, il n'est clairement pas désirable d'utiliser un *thread* natif par client. En effet, la manipulation de plusieurs milliers de *threads* natifs est lourde et le nombre de clients

[2] `http://www.jamendo.com`
[3] `http://www.blizzard.com`
[4] `http://osprey.ibiblio.org/`

servis aurait été limité par le nombre de *threads* actifs manipulables efficacement par le système ou par les librairies de *threads* en espace utilisateur (comme ST ou Pth).

Sans CPC, il faudrait utiliser une boucle à événements. Cependant, BitTorrent est un protocole asynchrone, car le pair distant maintient une liste de requêtes chez le *seeder*, et à état riche, car un pair peut être *choked* ou *unchoked* et *interested* ou *not-interested*. Ces deux caractéristiques rendent l'implémentation de BitTorrent avec ce genre de techniques compliquées et donnent des programmes difficiles à maintenir.

La façon le plus naturelle de gérer les entrées-sorties de chaque client serait de créer deux *threads par client* : un se chargeant d'analyser les demandes du client, et l'autre se chargeant de l'envoi des pièces pour lesquelles le pair distant maintient une liste de requêtes. CPC nous permet de créer ces deux *threads* par client sans se soucier de limiter le nombre de *threads* créés.

2.2 Montée en charge par rapport au nombre de torrents

Un *seeder* doit être capable de supporter un très grand nombre de torrents. Par exemple, Jamendo propose le téléchargement de ses 21 000 albums par BitTorrent[5] et on peut imaginer que la Bibliothèque Nationale de France voudrait mettre à disposition du public les 20 millions de documents du Catalogue collectif de France[6].

2.2.1 Utilisation de structures de données appropriées

Contrairement aux autres protocoles couramment utilisés, comme HTTP, qui ont toute l'information nécessaire encodée dans l'URL reçue, BitTorrent oblige le serveur à conserver en mémoire un certain nombre d'informations, les requêtes ne se faisant pas par URL mais à l'aide d'un *hash* précédemment calculé à et stocké dans un fichier (.torrent) de méta-données.

Pour supporter un grand nombre de torrents, il faut donc utiliser une structure appropriée pour conserver ces informations : l'identification des fichiers se faisant à l'aide d'un *hash*, l'utilisation d'une table de hachage est particulièrement appropriée.

Une entrée dans cette table de hachage ne prenant qu'une cinquantaine d'octets pour des torrents à un seul fichier et rarement plus d'une centaine pour des torrents multi-fichiers, l'utilisation mémoire reste très raisonnable même pour un très grand nombre de torrents (moins d'un méga-octet pour 10 000 torrents).

[5] Mais a des problèmes de *seed*

[6] S'il vous plaît, M. Frédéric Mitterrand.

2.2.2 Connection aux *trackers*

Les pairs doivent se connecter régulièrement aux *trackers*. Ces connections régulières posent un problème : si on sert 10 000 fichiers sur un *tracker* unique et que ce *tracker* demande au *seeder* une connection environ toutes les 30 minutes (les *trackers* que nous avons rencontrés demandent une connection toutes les 15 à 30 min) le *seeder* se retrouve à effectuer environ 6 connections par seconde.

On voit trois solutions a cette difficulté :

- se passer des connections aux *trackers* et transmettre les informations utiles au *tracker* par un moyen externe au protocole BitTorrent (ce que fait Permaseed), ce qui complique l'ajout de nouveaux *seeders* à l'essaim et force à utiliser un *tracker* spécifique ;
- utiliser des connections persistantes pour envoyer les requêtes aux *trackers* ;
- *pipeliner* les requêtes au maximum pour minimiser le nombre et le temps de connection aux *trackers*.

Notre implémentation actuelle tente de paralléliser au maximum les requêtes au tracker et permet de les désactiver totalement si besoin est. Par contre, le *pipelining* n'est pas encore implémenté.

2.3 *Memory mapping*

Le *seeder* passe la très grande majorité de son temps à lire des données sur le disque et à les écrire sur une *socket*. Il faut par conséquent bien choisir notre technique de lecture sur le disque.

Il existe à notre connaissance trois techniques possibles :

- les appels systèmes `read` et `write` ;
- le *memory mapping* (appel système *mmap*) ;
- l'appel système `sendfile`.

Les simples lectures dans un tampon puis écriture du tampon sur la *socket* nécessitent une copie supplémentaires (depuis le tampon en espace noyau vers le tampon en espace mémoire) et ajoutent une pression sur la mémoire. Cette méthode est donc peu adaptée à un *seeder* destiné à supporter une importante montée en charge.

Nous ne comprenons pas l'utilisation de *sendfile*, dont nous ne connaissions pas l'existence avant de choisir d'utiliser le *memory mapping*.

Nous avons choisi d'écrire sur la *socket* à partir de fichiers *mmapés*. Cette technique est simple à mettre en place. Contrairement à la combinaison d'appels à `read` et `write`, elle ne nécessite pas de tampon supplémentaire et est donc potentiellement plus efficace. Cette technique implique que la lecture sur le disque se fait au moment de l'appel à `write`.

2.4 Accès au disque

Le protocole BitTorrent implique que les pairs distants déterminent quelles parties des fichiers le *seeder* va avoir besoin de lire. En effet, il est conseillé de faire des requêtes de pièces aléatoires pour des raisons d'efficacité du protocole (expliquées dans la partie 3). Ceci entraîne une absence de localité d'accès au le disque.

Sans optimisations, ces lectures bloqueraient et entraîneraient un blocage total du programme.

Pour éviter ce problème il existe plusieurs possibilités :

- créer un *thread* natif par lecture, ce qui entraînerait rapidement un dépassement la *thread pool* ;
- utiliser des lectures asynchrones, peu pratiques sous Linux ;
- simuler des lectures non bloquantes avec *prefetch*.

Nous avons choisi d'utiliser des accès au disque avec *prefetch*.

Lorsque l'on reçoit une requête, on signale au noyau (à l'aide de l'appel système `madvise`) ce que l'on va avoir besoin de lire dans un futur proche. Le système se charge donc d'ordonner les accès aux disques comme il l'entend, ce qui nous décharge de la gestion des différents types de disques.

On attend ensuite que les données demandées soient chargées en cache pour les utiliser (on vérifie la présence dans le cache des données l'aide de l'appel système *mincore*). Si les données sont dans le cache lorsque vient le moment de les écrire, le *seeder* fait un appel à `write`. Dans le cas contraire, on place le *thread* CPC en fin de file (grâce à la primitive CPC `cpc_yield`) pour laisser plus de temps au noyau pour lire ces données.

Si les données ne sont toujours pas en cache lorsque le *thread* est exécuté de nouveau — ce qui s'est montré plutôt rare (environ un cas sur 10 lors de nos expérimentations à débit modéré) —, on dédie un *thread* natif à cette écriture.

3 Productivité du réseau

3.1 Éviter la famine sur certaines pièces

Pour éviter les famines, c'est-à-dire pour éviter que certaines pièces d'un torrent soient préférées aux autres, il faut choisir l'ordre dans lequel on choisit les pièces à demander ou servir. Si les demandes étaient faites de manière linéaire, il y aurait beaucoup plus de clients possédant les premières pièces d'un torrent et donc il y aurait une famine pour les dernières pièces.

La première méthode pour résoudre ce problème consiste à déléguer la question aux pairs distants en espérant qu'ils fassent des requêtes de manière totalement aléatoire, ce qui est suggéré par le protocole d'origine. Il existe une variante de cet algorithme, appelée l'algorithme *rarest first*, qui augmente les probabilités de choix pour les pièces les plus rares dans l'essaim.

Une deuxième méthode possible est l'algorithme de *super-seeding* : cet algorithme utilisé par le pair local et consiste à d'annoncer au pairs se connectant au *seeder* que l'on ne possède que les pièces que l'on veut lui envoyer. Cet algorithme (conçu par le créateur de BitTornado[7]) permet de répartir simplement les pièces dans l'essaim.

À l'heure actuelle notre *seeder* suppose que les demandes de pièces sont raisonnables et délègue donc la gestion de ce problème aux clients.

3.2 Équité

Pour maintenir une certaine équité entre les clients, on maintient deux ensembles disjoints de clients, un ensemble de clients que l'on sert (*unchoked*) et un ensemble de clients en attente (*choked*). À chaque client est associé un "crédit" qui diminue lors de l'envoi de données. Quand son crédit est épuisé, et selon la charge du serveur, soit on re-remplit ce crédit (cas de serveur peu chargé), soit on le place en attente (cas de charge moyenne), soit on le déconnecte pour laisser la place à un autre client.

En programmation sans variable de condition, implémenter l'équité de cette manière nécessiterait l'existence explicite de deux listes de pairs (servis ou en attente). Avec les variables de condition de CPC, ces listes sont créées implicitement : on met les *threads* responsables des pairs *choked* en attente sur une variable de condition commune, que l'on signale à chaque fois qu'un client se met en attente après avoir épuisé son crédit ou qu'un client se déconnecte.

4 Autres détails d'implémentation

Les points suivants sont moins essentiels que les précédents mais méritent d'être évoqués car l'utilisation de CPC permet de les résoudre différemment des techniques habituelles.

4.1 Communication entre *threads*

Nous utilisons plusieurs méthodes différentes pour la communication entre *threads*. La partie la plus intéressante de la communication entre *threads* concerne les *threads* d'envoi et de réception. On maintient une liste de pièces demandées par le pair distant et, à chaque fois que le *thread* chargé de l'envoi de ces pièces voit que cette liste est vide, il s'endort sur une variable de condition avec un temps d'attente limité à 30 secondes (ce qui est le seul cas de *timeout* dans notre implémentation). Cette variable sera signalée par le *thread* de lecture dès qu'il aura reçu une requête.

[7] BitTornado est disponible sur `http://bittornado.com`.

Quand le *thread* d'écriture est réveillé, soit la liste est encore vide — et dans ce cas on déconnecte le pair —, soit la liste a été remplie et dans ce cas le *thread* chargé de l'écriture reprend son exécution normale.

Il faut également synchroniser les *threads* de lecture et d'écriture d'un client au moment de sa déconnection pour éviter qu'un des deux tente d'avoir accès aux structures de données associées à ce client après leur libération. Pour cela, on utilise une barrière, issue de la librairie standard de CPC.

4.2 Autres détails

Pour éviter de faire de nombreux `read` (appel système, donc coûteux), on utilise un tampon et à chaque fois que ce tampon ne contient pas un message complet on tente de le remplir et tant qu'on n'a pas lu un minimum donné, on laisse les autres *threads* s'exécuter.

Étant donné que l'appel système `write` est aussi coûteux que `read`, on tente d'optimiser le `write` aussi, nous utilisons l'appel système `writev` qui nous permet de faire une écriture à partir de plusieurs tampons en un seul appel système au lieu de faire un appel système `write` par tampon, ce qui est plus efficace et évite les délais dûs à l'utilisation de l'algorithme de Nagle.

5 Problèmes ouverts

Il manque encore à notre *seeder* un certain nombre de fonctionnalités que nous n'avons pas eu le temps d'implémenter ou pour lesquelles nous n'avons pas trouvé de solution élégante.

5.1 Mise-à-jour des fichiers servis

Un serveur est sensé ne jamais s'arrêter. On voudrait donc que le *seeder* vérifie régulièrement l'ajout et la suppression de torrents à servir, ainsi que la mise à jour des autres informations les concernant.

Actuellement, il faut redémarrer notre programme si on désire ajouter, supprimer ou modifier des fichiers servis. De plus, on ne vérifie jamais si les méta-données correspondent bien aux fichiers servis.

La solution qu'on envisage serait de créer un *thread* — probablement un *thread* détaché (natif) puisqu'il nécessite de nombreux accès au disque, dont le rôle serait de vérifier régulièrement l'état de l'arborescence des fichiers servis par rapport aux torrents présents dans notre table de hachage et réagir à chaque changement :

- pour un ajout, on l'ajoute dans notre table de hachage et on attend la prochaine série d'annonces aux *trackers* pour profiter des connections persistantes ;
- pour une suppression, on retire le torrent correspondant de la table de hachage et annoncer au *tracker* qu'on ne sert plus ce torrent. Il ne

faudra pas omettre de déconnecter les pairs connectés pour ce torrent ni d'*unmapper* les données du torrent.

5.2 *Timeout*

Il se peut qu'un client soit connecté au *seeder* et ne fasse aucune requête (par exemple un client incorrect ayant le torrent complet). Il se peut également qu'un client soit déconnecté sans avoir interrompu correctement la connection TCP. Dans ces cas, il faut déconnecter ce client puisqu'il occupe des ressources inutiles, en monopolisant principalement un descripteur de fichiers et un peu de mémoire physique. Pour détecter des pairs de ce genre, on a besoin de *timeouts* de couche application car le *timeout* de couche réseau ne suffist pas forcément.

Une solution simple consiste à attacher un *timeout* explicite à chaque opération d'entrée/sortie. Cette solution entraîne l'ajout de tests pour vérifier l'état de chacune de ces opérations, ce qui compliquerait le code.

Une solution plus modulaire serait désirable, par exemple avoir un *thread* encapsulé pour chaque *timeout* comme c'est la cas pour les barrières. Pour le moment nous n'avons pas trouvé de solution satisfaisante pour résoudre cette difficulté.

6 Conclusion

Notre but était d'écrire un programme consistant en CPC, ce qui n'avait jamais été fait auparavant. Le *seeder* que nous avons réalisé fonctionne et présente de très bonnes performances dans les conditions de test que nous avons pu mettre en place.

Deux problèmes restent ouverts : la gestion modulaire des *timeouts* et la mise en place d'un protocole de tests permettant de réaliser des mesures réalistes de performances.

A Disponibilité du logiciel

Notre logiciel est disponible, sous licence libre (licence MIT) à l'adresse suivante : `http://www.pps.jussieu.fr/~jch/software/hekate`.

Références

[1] Gabriel Kerneis, *CPC, des threads coopératifs par passage de continuation. Rapport de stage de master 2.* juin 2009.

[2] Juliusz Chroboczek, *Continuation Passing for C : A space-efficient implementation of concurrency.* juin 2005.

[3] Juliusz Chroboczek, *The CPC manual.* 2004–2007.

[4] C.A.R. Hoare, *Towards a theory of parallel programming. In Operathig Systems Techniques*, Academic Press, New York, 1972.

[5] Boussinot, F. , *FairThreads : Mixing Cooperative and Preemptive Threads in C – Concurrency and Computation : Practice and Experience*, 2005.