TP 8 : Multiplexage des communications réseau
Implémentation avec *Madeleine*

Alexandre Denis
`denis@labri.fr`

Systèmes et supports d'exécution — 2004–2005

L'objectif de ce TP est de d'implémenter un mécanisme de *multiplexage* des communications réseau permettant à plusieurs *threads* de communicaquer simultanément sur un seul canal partagé.

Ce TP est **noté** et à rendre (code source + compte-rendu) par e-mail au plus tard le **jeudi 13 janvier 2005**.

1 La bibliothèque *Madeleine*

1.1 Principe

Madeleine est une bibliothèque de communication portable et générique, capable d'utiliser efficacement des réseaux haute performance tels que *Myrinet* ou SCI. Une des originalité de *Madeleine* est que l'interface est construite à l'aide de contraintes exprimées au niveau *sémantique* (« *cheaper* », « *express* », etc.) et non de manière impérative (« *en-tête* », « *corps de message* », etc.) comme la plupart des autres bibliothèques de communication.

Le modèle proposé repose sur la notion de *canaux* qui, de façon semblable aux *communicateurs* MPI, sont composés d'un ensemble de nœuds et permettent les communications de chacun des nœuds vers chaque autre nœud du canal. À l'intérieur d'un canal, les nœuds sont identifiés par un numéro logique ; la numérotation commence à 0. Dans ce TP, nous n'utiliserons qu'un seul canal.

Madeleine est développée dans l'équipe de recherche *Runtime* du *LaBRI*. Vous trouverez des informations complémentaires à l'adresse :
`http://dept-info.labri.fr/~aumage/Runtime/logiciels.html`

1.2 Interface de programmation

L'échange de messages est effectué à l'aide d'une interface de construction de messages *incrémentale* :

`mad_begin_packing(canal, destinataire)` débute un envoi de message sur le canal et vers le destinataire donné. La fonction renvoie un objet *connexion*.

`mad_pack(connexion, buffer, size, mode_envoi, mode_recéption)` ajoute un paquet à un message commencé par un `mad_begin_packing`.

`mad_end_packing(connexion)` termine un message.

Des contraintes sémantiques sont données pour préciser *quand* Madeleine accède aux buffers, en donnant un mode d'émission et un mode de réception pour chaque paquet. Le mode d'émission détermine les garanties données à l'émetteur ; le mode de réception détermine les garanties côté récepteur.

mode d'émission	*sémantique*
`mad_send_SAFER`	la valeur envoyée est celle au moment du `mad_pack`
`mad_send_LATER`	la valeur envoyée est celle au moment du `mad_end_packing`
`mad_send_CHEAPER`	aucune garantie avant le `mad_end_packing`

mode de réception	*sémantique*
`mad_receive_EXPRESS`	les données sont disponibles immédiatement après le `mad_unpack`
`mad_receive_CHEAPER`	aucune garantie avant le `mad_end_unpacking`

Il va de soit que les modes « *cheaper* » pour lesquels il est interdit de toucher aux données entre le `begin_packing` et le `end_packing` autorisent de meilleures performances, car *Madeleine* peut alors réordonnancer ou agréger les paquets, mais ces modes ne conviennt pas à tous les cas.

Chaque primitive d'émission à son symétrique pour la réception. On utilisra ainsi :

`mad_begin_unpacking(canal)` pour attendre un nouveau message. *Note* : cette fonction est bloquante et ne permet pas de choisir la source du message — on est obligé de traiter le message qui arrive, même si ce n'est pas celui qui était attendu.

`mad_unpack(connexion, buffer, size, mode_envoi, mode_recéption)` reçoit un paquet

`mad_end_unpacking(connexion)` termine la réception du message

Attention, il est primordial que les `mad_pack` et `mad_unpack` soient symétriques : même taille, même modes. En effet, *Madeleine* n'envoie pas d'en-tête[1] sur le réseau donc ne peut pas retrouver les frontières de paquets sans votre aide.

1.3 Exemple

Par exemple, pour envoyer des données stockées dans `buffer` de taille `size` :

```
p_mad_connection_t connection = mad_begin_packing(channel, dest);
mad_pack(connection, buffer, size, mad_send_CHEAPER, mad_receive_CHEAPER);
mad_end_packing(connection);
```

Et côté récepteur :

```
p_mad_connection_t connection = mad_begin_unpacking(channel);
mad_unpack(connection, buffer, size, mad_send_CHEAPER, mad_receive_CHEAPER);
mad_end_unpacking(connection);
```

1.4 Initialisation

Reprendre tel-quel le code d'initialisation donné sur la page web du cours SSECPD. **Attention** : nous utilisons des *threads Marcel*. Le code d'initialisation est légèrement différent de celui que nous avions utilisé avec *Madeleine* sans *Marcel*.

[1] À votre avis, pourquoi ?

2 Travail à faire

2.1 Benchmark de référence

Nous commençons par écrire un programme de banchmark qui servira de référence dans la suite. Commencez par télécharger le fichier `timing.h` qui contient les primitives de mesure de temps.

Le banchmark doit avoir les propriétés suivantes :
- être en *ping-pong* (alternativement un message dans un sens, un message dans l'autre sens). Pourquoi est-ce meilleur qu'une succession d'envois dans le même sens ? Que risquerait-on de mesurer sinon ?
- mesurer les performances pour des messages de taille variable — faire les puissances de 2 en ordre croissant, de 4 octets à 4 Mo.
- réaliser une *moyenne* de plusieurs allers-retours pour chaque taille, pour une meilleure précision. Choisissez et justifiez le nombre d'allers-retours qui vous semble adéquat.

Tracer ensuite la courbe de débit et la courbe de temps de transfert en fonction de la taille des messages à l'aide de `gnuplot` (ou de tout autre outil de tracé de courbes). Pour la courbe de débit, choisissez une échelle logarithmique en x, pour la courbe de temps une échelle logarithmique en x et en y. N'hésitez pas à zoomer uniquement sur les petites tailles de message pour la courbe de temps.

2.2 Multiplexage

2.2.1 Communications avec plusieurs *threads*

On désire maintenant faire communiquer plusieurs *threads* en même temps.
- Réalisez un petit programme qui fait un ping-pong simple avec un jeton de taille déterminée.
- Ajoutez un deuxième *thread* qui fait circuler son propre jeton. Que se passe-t-il si les deux jetons ne font pas la même taille ? Essayez, pour voir ! Comment expliquez-vous le résultat obtenu ?
- Réfléchissez à un mécanisme qui permettrait d'assurer que chaque *thread* reçoit son jeton et pas le jeton destiné à l'autre *thread*.

L'idée naturelle serait d'utiliser des canaux Madeleine distincts. Malheureusement, Madeleine fournit un nombre de canaux limité par ce que le matériel est capable d'offrir : deux canaux sur Myrinet, un seul canal sur SCI. Puisque nous voulons un nombre arbitraire *de canaux, cette solution n'est pas satisfaisante. Dans la suite, vous allez concevoir un mécanisme capable d'offrir un nombre quelconque de canaux indépendants au-dessus de Madeleine.*

2.2.2 Sous-canaux

Nous allons écrire une surcouche basée sur *Madeleine* que l'application utilisera à la place de *Madeleine*. Cette surcouche offrira un nombre de canaux (ou plutôt *sous-canaux*, pour ne pas confondre avec les canaux *Madeleine*) décidé par l'application. Nous proposons d'écrire les fonctions suivantes :

`sc_init(int sub_chan_max)`
: Fonction d'initialisation appelée au début du programme (devra appeler `common_init`) qui permet d'utiliser par la suite `nb_sub_max` sous-canaux.

`sc_end()`
: Fonction de terminaison qui doit être appelée par le programme avant sa terminaison.

`sc_begin_packing(int sub_chan, ...)`
: Commence un envoi sur un sous-canal ;

`sc_pack(int sub_chan, ...)`
: Envoie une partie des données d'un message sur le sous-canal ;

`sc_end_packing(int sub_chan, ...)`
: Termine un envoi sur un sous-canal ;

`sc_begin_unpacking(int sub_chan, ...)`
: Commence une réception sur un sous-canal ;

`sc_unpack(int sub_chan, ...)`
: Reçoit une partie des données d'un message sur le sous-canal ;

`sc_end_unpacking(int sub_chan, ...)`
: Termine une réception sur un sous-canal ;

- Réfléchissez aux prototypes complets des fonctions proposées ci-dessus. Vous pouvez en ajouter d'autres.
- Proposez une implémentation de ces fonctions sans vous soucier des performances pour l'instant.
- Qui doit faire le `mad_begin_unpacking` ? Pourquoi ? Quels sont les choix possibles ? Quel choix avez-vous fait ?
- Modifiez votre programme du 2.2.1 pour utiliser ces nouvelles fonctions plutôt que *Madeleine*. Vérifiez que, cette fois, les jetons ne sont plus mélangés !

2.2.3 Performances et optimisation du multiplexage

Nous nous soucions maintenant du surcoût introduit par le multiplexage.
- Quelle est l'impact de votre version de multiplexage sur les performances ? (copies, en-têtes, ...)
- Vérifiez les performances en modifiant le programme de benchmark du 2.1 de façon à lui faire utiliser les nouvelles fonctions plutôt que *Madeleine* directement.
- Quelle optimisation proposez-vous ? Est-ce possible en gardant la même interface de programmation ou faut-il la modifier ? Justifiez.
- Modifiez votre implémentation pour prendre en compte vos optimisations, ré-essayez le benchmark, et comparez aux résultats obtenus sans multiplexage. Quel est le surcoût, au final ?
- Le multiplexage introduit-il forcémment un surcoût ou est-il possible de le réaliser « quasi-gratuitement » ? Justifiez.