MIF32 - Compte-Rendu du projet CAN

Adrien FAURE(p1006169) & Raphaël CAZENAVE-LEVEQUE (p1410942)

2014-2015

Abstract

Nous présentons une implémentation en C/MPI d'une DHT utilisant le protocole CAN. Nos choix d'implémentation permettent une recherche efficace dans la DHT depuis n'importe quel noeud. Nous offrons aussi des représentations graphiques en SVG et textuelles de l'état de la DHT ainsi qu'une couverture de nos fonctions les plus critiques grâce à des tests unitaires. Ces représentations sont générées automatiquements lors de l'exécution de la DHT, ou à la demande si l'on utilise l'interface en ligne de commande. URL du projet : *https://bitbucket.org/OldDadou/can*

1 Fonctionalités disponibles

Les cinq étapes décrites dans le sujet on été respectées. A l'exception de la suppression des noeuds, jugée trop complexe lors des séances de TP. Nous proposons cependant une solution à la fin de ce rapport.

1.1 Etape 1 : Création du réseau et insertion de noeud

Lors de l'exécution de notre programme, le nombre de processus est statique, défini lors de l'appel à la commande *mpirun*. Le rang du processus root est défini par une macro. Celui-çi nous servira de coordinateur, et ne fera pas partie de la DHT. Le coordinateur demandera successivement aux noeuds de s'insérer. Lors d'une insertion, l'algorithme décrit dans le sujet a été respecté, de plus, si l'ancien noeud contient des données, il les transmettra au nouvel arrivant. Le nouveau noeud se verra recevoir son voisinage communiqué par l'ancien noeud. De plus celui-çi enverra à chacun de ses nouveaux voisins sa nouvelle frontière afin qu'ils se mettent à jours.

1.2 Etape 2 : Insertion de données

Via le coordinateur (ou depuis n'importe quel noeud, car les méthodes sont génériques), il est possible d'insérer des données à une position *(x, y)* dans l'espace cartésien. Même si dans notre projet, nous n'insérons que des int, l'implémentation sous-jacente permet de définir le type des données à insérer via un système de TAG et en spécifiant la taille. Les données seront acheminées, via notre algorithme de routage, au sein de l'overlay, sous forme d'un tableau d'octet.

1.3 Etape 3 : Recherche et lecture de données

Pour effectuer la récupération d'une donnée, une requête contenant le rank du demandeur et les coordonées seront envoyées dans l'overlay et acheminée jusqu'au noeud responsable de la zone correspondante. Le noeud responsable renverra, si elle existe, la donnée ainsi que les métadonnées fournies (type et taille) au demandeur. Il sera alors de la responsabilité de l'utilisateur de les ré-encoder lors de la réception grace aux TAG et en utilisant la taille dans le cas d'un tableau par exemple.

1.4 Etape 4 : Suppression de noeud

Suite au retrait cette étape, nous proposons la solution suivante, que nous n'avons pas implémentée. Lors de la suppression d'un noeud, le noeud en question emmettra une requeête à ses voisins pour déterminer le plus apte à gérer une autre zone. Le noeud nouvellement en charge ne fuisionnera pas les zones, appliquera la logique de notre DHT sur chacunes de ces zones. Il serait alors possible d'exécuter de temps en temps un algorithme de ré-homogénéisation des zones de la DHT. Il est amusant de constater qu'un même processus pourrait être en charge de zones non-juxtaposées.

1.5 Etape 5 : Suppression de données

La suppression d'une donnée est similaire à une recheche, si ce n'est qu'une fois acheminée, celle-çi sera supprimée.

2 Choix de conception

2.1 Architecture

Notre projet propose l'architecture suivante. Une partie correspondant à l'interface de la DHT. La suppression et l'ajout d'une donnée. Une partie correspondant à la logique de la DHT, elle même séparée en deux sous-parties. L'une est dédiée à la coordination (ajout, suppression de noeuds) et utilisée par le processus coordinateur. Cependant son role est superficiel, car envoit l'ordre aux noeuds d'initier la procédure d'insertion dans l'overlay. Le noeud ainsi sommé demandera un point d'entré, et une fois celui-çi obtenu, il emettra une requete au point d'entré obtenur pour rejoindre le réseau. Cette procédure permettrait de s'emanciper totalement du noeud coordinateur pour s'insérer dans l'overlay. L'autre sous-partie est l'ensemble des traitements associés à une requête émise par les noeuds de l'overlay. La dernière partie de notre architecture est composée de toute la logique de manipulation des espaces cartésiens abtrait de l'utilisation d'MPI. Cette partie étant source d'erreur, elle est en majeure partie couverte par des tests unitaires.

2.2 Communications

Les processus non-coordinateurs seront en attente, en premier lieu de l'ordre de s'insérer du coordinateur. Ils attendent ensuite de recevoir et de traiter une requête. Les noeuds seront en mesure, lors d'une réception, de traiter les requêtes grâce au système de tag proposé par MPI.

2.3 Journalisation et debug

La complexité du code augmentant, nous avons mis en place plusieurs systèmes afin de comprendre le comportement de notre logiciel. En premier lieu, nous avons mis en place la journalisation des manipulations de l'overlay via le coordinateur. Le coordinateur dispose de routine afin de demander des informations aux noeuds de l'overlay. Les informations récupérées sont stockées dans un répertoire de logs sous deux formes. Une forme textuelle, et une forme graphique au format SVG. Nous utilisions beaucoup l'"aléatoire", la reproductibilité était donc impossible, donc nous avons mis en place un système de seed paramétrable. Celle-çi est automatiquement journalisée dans le fichier *logs/global.txt* et peut être spécifiée lors de l'exécution du programme. De plus nous pouvons spécifier que nous souhaitons lancer le programme et manipuler la DHT grace à un prompt.

3 Les structures de données mises en oeuvre

Chaque noeud possède localement plusieurs données. Une liste de frontières représentant le voisinage proche du noeud, ainsi qu'une liste capable de stocker les données dont le noeud est responsable. L'espace dont le noeud est reponsable est défini par une structure représentant un rectangle.

4 Algorithme de routage

Afin de localiser un noeud dans l'overlay, nous envoyons une requete composée de la source (*comm_ rank* de l'initiateur) ainsi que de la destination (coordonnées x,y). la procédure se déroule en deux étapes. Le noeud récepteur vérifiera si les coordonnées appartiennent à la zone dont il est responsable, si c'est le cas, il sera alors en mesure de répondre à la requête grâce au rang de l'initiateur présent dans la requête. Autrement si les coordonnées fournis n'appartiennent pas à la zone du noeud, alors il déterminera, parmis ses voisins, le prochain saut. Cet algorithme, en apparence simple, n'a été possible que grâce au maintient permanent de la cohérence de la DHT. En effet, lors de l'insertion d'un nouveau noeud, il est primordial de mettre à jour le voisinage des noeuds, ceci à nécessité une bonne maitrise des communications.

5 Pour aller plus loin

5.1 Optimisation

Les communications MPI utilisant de l'attente active. Nous utilisons la fonction *MPI_ Iprobe* couplée à un *usleep* afin de réduire la consommation de ressource engendrée par cette attente active.

5.2 Ajout dynamique de processus lors de l'exécution

Nous avons souhaité ajouter la possibilité d'ajouter des processus au cours de l'exécution. Nous avons effectué des essais en utilisant *MPI_ Spawn*, puis *MPI_ Intercomm_ merge* pour obtenir un communicateur intra contenant tous les processus ainsi que le nouveaux processus. Malheureusement, il n'est pas possible de re-merger ce communicateur [1] lors de spawn successifs, car *MPI_ Intercomm_ merge* est une routine collective et donc chaque processus précédément spawné doit participer au prochain merge.

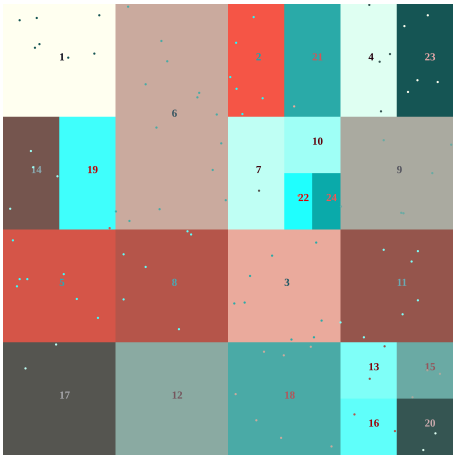

Figure 1: 25 Processus - 100 datas

[1] Nous avons du abandonner cette idée suite à la lecture de ce topic de la mailling-list d'openMPI - `https://www.open-mpi.org/community/lists/users/2007/10/4312.php`

A Annexe 1 - README

```
# pour builder:

    make


# Pour builder et executer les tests (nécessite la bibliothèque cunit)

    make runtest


# Pour builder les test sans executer les tests (nécessite la bibliothèque cunit)

    make test


# pour builder l'appli sans executer les tests  :'(

    make mpi_can

# execution avec prompt :

    mpirun -np 7 ./mpi_can debug


# execution sans prompt :

    mpirun -np 7 ./mpi_can


# Execution with yield
    mpirun --mca mpi_yield_when_idle 1 -np 10 mpi_can

# execution avec prompt et en fixant une seed spécifique au générateur pseudo-aléatoire pour pouvoir reproduire les résultats (et les bugs ^^):

    mpirun -np 7 ./mpi_can seed 42 debug


# execution sans prompt et en fixant une seed spécifique au générateur pseudo-aléatoire pour pouvoir reproduire les résultats (et les bugs ^^)::

    mpirun -np 7 ./mpi_can seed 42


# Prompt:

    -- 9 nodes availables --
    > status              : show log about the state of the DHT
    > insert 2            : insert the node 2 in the overlay
```

```
48      > insert all          : insert all nodes in the overlay and run etape 3
49      > log                 : add a textual/SVG log on logs/ directory
50      > put <x> <y> <data> : put <data> in position (x, y)
51      > get <x> <y>         : get a data from position (x, y)
52      > shuffle <nb>        : randomly insert <nb> random data
53      > rm <x> <y>          : remove a data in position (x, y)
54      > etape3 <x>          : insert <x> element and try to retrieve the 5
            last and first
```