Système de gestion de données distribué

Rapport de projet

Cours de systèmes distribués
M1 Informatique Fondamentale
ENS de Lyon
Printemps 2018

guillaume.coiffier@ens-lyon.fr

L'intégralité du code source est disponible à l'adresse suivante :
https://github.com/GCoiffier/Distributed-Data-Manager-Erlang

Table des matières

1 Description de la solution proposée **1**
1.1 Topologie du réseau 1
1.2 Initialisation du serveur 1
1.3 Méthode de stockage 1

2 Fonctionnement du programme **1**
2.1 Interface client 1
2.2 Les requêtes *send* 2
2.3 Les requêtes *retrieve* 2
2.4 Ajout et suppression de noeuds de requête 3
2.5 Election de leader et maintient de l'agent **master** 3

3 Discussion sur les performances et la robustesse de la solution choisie **3**
3.1 Robustesse du réseau 3
3.2 Equilibre de la charge en données 4
3.3 Complexité de communication 4

1 Description de la solution proposée

La solution que nous proposons dans ce projet a été inspirée du fonctionnement du middleware DIET [1] pour ce qui est de la classification des agents en plusieurs tâches et catégories. La topologie est cependant complétement différente.

1.1 Topologie du réseau

Nous avons fait le choix d'imposer une topologie précise à notre réseau, plutôt que de gérer une topologie arbitraire. Les conséquences de ce choix sont discutées en section 3.
On distinguera, dans le réseau, trois types d'agents :

— L'agent **master**, responsable des connections entrantes au réseau.

— Les agents de requête (**query**), responsables de la gestion des requêtes de stockage et d'accès aux données.

— Les agents de stockage (**storage**) responsables... du stockage des données.

Le noeud **master** est relié à l'intégralité des noeuds **query**. Ces derniers forment une clique. Dans la pratique, ils ont simplement accès à la liste des PiD de tous les autres noeuds query. Enfin, les noeuds de stockages sont reliés à un ou plusieurs noeuds **query** et forment leurs fils.

1.2 Initialisation du serveur

L'agent **master** (dont le code se situe dans le module `server.erl`) est responsable de l'initialisation du serveur. Il est lancé lorsque l'on compile le module `server`. Il a la tâche d'initialiser N noeuds de requêtes, qui vont à leur tour initialiser M noeuds de stockage chacun.
Les valeurs par défaut de N et M sont respectivement 10 et 5.

1.3 Méthode de stockage

Le stockage des données est assuré par les agents de stockages. Dans la pratique, ces agents disposent d'une structure de données de type dictionnaire, associant, à un identifiant unique, un morceau de données. Il est possible de communiquer avec ces agents pour leur envoyer des données à stocker, et pour récupérer les données associées à un identifiant.

2 Fonctionnement du programme

2.1 Interface client

L'interface du client est décrite par le module `client.erl`. Elle comprend les fonctions suivantes :

— `connect/1`
— `disconnect/0`
— `update_connection/0`
— `send_data/2`
— `fetch_data/1`
— `release_data/1`
— `get_stored/0`
— `add_to_node/2`
— `kill_process/1`

Grâce à cette interface, on peut communiquer avec le serveur de façon simple. Les fonctions `client:connect/1`, `client:disconnect/0` et `client:update_connection/1` servent pour la connexion au serveur.

Dans la pratique,la fonction `client:connect/1` prend en argument le nom du noeud sur lequel tourne l'agent **master**. Elle demande à cet agent la liste des agents de requêtes disponibles sur le réseau, et stocke cette liste dans un petit processus annexe (consistant,

1. https://graal.ens-lyon.fr/diet/

comme pour les agents de stockages, en une boucle sur une structure de *set* avec laquelle on peut communiquer).

À chaque requête effectuée sur le réseau, le client choisit au hasard parmi les agents de requêtes dont il dispose et envoie sa requête à l'agent choisi.

La fonction `client:disconnect/0` réinitialise le client.

La fonction `client:update_connection/1` est similaire à la fonction `client:connect/1` sauf qu'elle suppose une initialisation du client et de la connexion.

2.2 Les requêtes *send*

Les requêtes *send* sont les requêtes demandant le stockage d'une donnée particulière sur le réseau. Elles sont gérées par la fonction `client:send_data/2`. Cette fonction prend en argument le nom du fichier à envoyer ainsi qu'un mode de stockage.

Le fichier à envoyer doit se trouver dans le sous dossier *data*. Aucune hypothèse n'est faite sur son extension (et nous pouvons constater qu'il est tout à fait possible d'envoyer et de récupérer des extensions complexes telles que le .odt)

Le mode de stockage est un *atom*. Quatre modes sont disponibles :

simple : le fichier est envoyé en entier sur un agent de stockage. Pas de redondance

critical : le fichier est dupliqué et envoyé en entier sur l'intégralité des agents de stockage d'un certain agent de requête. Redondance forte.

distributed : le fichier est découpé en morceaux, et chaque morceau est stocké sur l'un des agents de stockage d'un certain agent de requête.

hybrid : le fichier est découpé en morceaux, et stockés en double sur les fils de deux agents de requêtes. Ce mode de stockage introduit également de la redondance, mais à moins grande échelle que le mode *critical*.

Une requête *send* est envoyée par un client à n'importe quel agent de requête du serveur. Cet agent, noté $\mathcal{A}$, sera responsable du stockage des données. La procédure varie en fonction du mode de stockage :

simple : $\mathcal{A}$ choisit, parmi les agents de stockages dont il a accès, celui qui stocke le moins de données, et lui demande de stocker cette nouvelle donnée.

critical : $\mathcal{A}$ envoie la donnée à chacun de ses agents de stockage.

distributed : $\mathcal{A}$ découpe la donnée en autant de morceaux qu'il a d'agents de stockage, puis distribue les données.

hybrid : $\mathcal{A}$ effectue les mêmes opérations que pour *distributed*, puis choisit un autre agent de requête $\mathcal{B}$ au hasard et lui demande de stocker la donnée en mode *distributed*.

Avant d'envoyer les données à leur agents de stockage, les agents de requêtes créent un identifiant pour retrouver la donnée plus tard. Cet identifiant consiste en un tuple contenant le nom du fichier de données, la date (an/mois/jour/heure/minute/seconde en temps universel) ainsi qu'un nombre aléatoire entre 0 et 1 000 000 000.

Lorsque la donnée a bien été stockée, le noeud de requête renvoie une confirmation au client, contenant le mode de stockage ainsi que l'identifiant de la donnée. Le client stocke ces informations dans un dictionnaire.

2.3 Les requêtes *retrieve*

Les requêtes *retrieve* sont de deux types :

- Les requêtes qui demandent des données sans les supprimer du serveur (`client:fetch_data/1`)
- Les requêtes qui récupèrent les données du serveur en les supprimant (`client:release_data/1`)

La différence entre ces deux requêtes s'effectue chez les agents de stockage, qui suppriment ou non les données dont ils disposent.

Les deux fonctions `client:fetch_data/1` et `client:release_data/1` prennent en argument le nom du fichier à récupérer. Grâce à ce nom, le client peut chercher dans son dictionnaire local l'identifiant unique sous lequel est stocké le fichier sur le serveur. Cet identifiant est ensuite envoyé à un agent de requête aléatoire.

Côté serveur, lors de la reception d'une requête *retrieve* par un agent de requête, ce dernier demande à tous ses noeuds de stockage si une telle donnée n'existe pas chez eux, et broadcast la requête chez tous les autres agents de requête. Ainsi, tous les agents de stockage sont interrogés, et les autres agents de requêtes ne répondent que s'ils ont une réponse positive.

L'agent qui a été interrogé par le client récupère donc toutes les réponses :

simple : il reçoit un seul élément contenant l'intégralité de la donnée et le renvoie au client

critical : il reçoit un certain nombre de copies de la donnée et en renvoie une au client

distributed : il reçoit les morceaux de la donnée, les remet dans l'ordre, les fusionne, et les renvoie au client.

hybrid : il reçoit en double les morceaux de la donnée, se débarrasse des doublons, fusionne et renvoie le reste.

Les données récupérées par le client sont écrites dans le dossier *output*

2.4 Ajout et suppression de noeuds de requête

Grâce aux fonctions`add_to_node/2` et `kill_process/1`, un client peut ajouter et supprimer des agents sur le serveur.

Lors de l'ajout d'un agent, on prend en paramètre le noeud sur lequel tourne le master, et un autre noeud (potentiellement le même) sur lequel ajouter l'agent. Cet agent rejoins automatiquement la clique des agents de requête, et crées ses propres agents de stockage sur le même noeud.

Lors de la suppression d'un agent de requête, afin de ne pas perdre de données, ses agents de stockages sont cédées chacun à un autre agent de requête tiré au hasard, qui en devient donc responsable.

Il n'est pas possible de supprimer directement un agent de stockage.

2.5 Election de leader et maintient de l'agent master

Un protocole d'élection de leader a été mis en place entre les agents de requête. Le leader a la responsabilité d'envoyer des pings au master. Dans le cas d'une absence de réponse de la part du master, le leader spawn un nouveau master.

Le protocole d'élection est initié par le leader, lorsque le résultat d'un générateur pseudo-aléatoire prenant des valeurs de 1 à *freq_leader_election*, relancé toutes les secondes, vaut 1. Dans la pratique, *freq_leader_election* vaut 50.

3 Discussion sur les performances et la robustesse de la solution choisie

3.1 Robustesse du réseau

La structure de clique est la plus solide possible, mais fait intervenir beaucoup de liens. Chaque agent de requête doit, en effet, stocker l'intégralité des PiDs des autres agents de requêtes.

De plus, la connexion au reste du réseau des agents de stockage est unique, et pourrait venir à être coupée dans le cas du crash de leur agent de requête. Dans un premier temps, nous aavons envisagé de ne pas dissocier un agent de requête de ses agents de stockage. Ainsi, la perte de l'agent de requête entrainait la perte pure et simple des données. Nous avions l'intention de faire en sorte qu'un agent de stockage ait plusieurs 'pères', permettant une meilleure solidité du réseau. Néanmoins, cette solution s'est avérée problématique dans le protocole de récupération des données, car cela multipliait les messages échangés.

Nous avons donc opté pour une deuxième solution : un agent de stockage envoie régulièrement des pings à son père. Lorsque ce dernier ne répond pas, il demande à l'agent master un nouvel agent de requête à qui se connecter.

3.2 Equilibre de la charge en données

L'équilibre de la charge en données est uniquement géré à l'arrivé de nouvelles données. Lors d'un ajout de données simple, l'agent de requête selectionne son agent de stockage qui a le moins de données stockées. L'agent de requête garde toujours en mémoire un dictionnaire donnant la charge de ses agents de stockage.

Les données peuvent être considérées comme bien distribuées sur le réseau, car un client demande toujours un agent au hasard lors de ses requêtes.

Aucun mécanisme d'équilibrage supplémentaire n'a cependant été implémenté.

Une idée serait de donner aux agents de requête la possibilité de créer de nouveaux agents de stockage lorsque ceux qu'ils possèdent atteignent une charge critique, puis d'implémenter un mécanisme de passation de ces agents de stockage à d'autres agents de requête afin d'en équilibrer leur nombre.

3.3 Complexité de communication

Les requêtes *send* sont relativement peu gourmandes en messages. Il s'agit juste pour le client d'envoyer un unique message à un agent de requête. Ce dernier ne communiquera ensuite qu'avec ses agents de stockage (ou avec un seul autre agent de requête dans le stockage hybrid).

Les requêtes *retrieve*, en revanche, touchent l'intégralité du réseau. Un nombre linéaire (en le nombre d'agents du réseau) de messages est envoyé pour chaque requête, ce qui peut être un problème lors du passage à l'échelle. Etant donné la façon dont les données sont distribuées, et que les agents de stockages sont susceptibles de changer de père, il n'est pas possible de faire autrement si l'on veut être sûr de tout bien récupérer.

Des heuristiques peuvent cependant être mises en place : on peut demander aux agents de requête un par un, au lieu de tous en même temps, et s'arrêter dès que l'on a reconstitué les données. En revanche, tous les agents seront toujours interrogés dans le pire cas.

Réponse aux questions

Nous détaillons ici les solutions apportées (ou non) aux différentes questions de l'énoncé

Question 1 : topologie

a) *Spawn an agent* : **OK**

b) *Have this agent be part of a coherent topology* : **OK**

c) *Be able to have a new agent join the topology* : **OK**. Il est possible d'ajouter un agent en utilisant la fonction `client:add_to_node/2` (cette fonction prend deux noeuds en paramètre, le noeud où doit se situer le **master**, et le noeud sur lequel créer le nouvel agent).

d) *Remotely kill an agent* : **OK**. Il est possible de tuer un agent en utilisant la fonction `client:kill_process/0` qui tue un agent au hasard (utile pour les tests) ou la fonction `client:kill_process/1`, qui tue l'agent dont le PiD a été donné en argument.

e) *Have the possibility to detect and react to an agent dying without prior notice* : on ne réagit qu'à la mort de l'agent **master**. Les crashs des autres agents ne sont pas gérés.

Question 2 : communication

a) *Broadcast* : **OK** (fonction `client:broadcast/1`)

b) *Scatter* : **OK** (fonction `client:scatter/1`)

c) *Leader election* : **OK**. Utilisée pour pouvoir se maintenir au courant de l'état de l'agent **master**.

Question 3 : gestion de données

a) *Receive pieces of information, remember them and return a UUID* : **OK**. Voir la section 2.2

b) *Your agents should be able to send back the piece of data corresponding to a UUID* : **OK**. Voir la section 2.3

c) *Make sure that no data is lost when a nodes is asked to shut down* : **OK**. Voir la section 2.4

d) *Have a mechanism to release a specific data (deleting it from all nodes where it is stored) ;* : **OK**. Nous avons implémenté deux façons de récupérer les données, `client:fetch_data/1` et `client:release_data/1`. La seconde fonction efface les données du réseau.

e) *Have a mechanism to distribute all the pieces of data evenly across the platform so that no node store them all* : ∼**OK**. La distribution s'effectue par la sélection aléatoire de l'agent de stockage par le client. Aucun mécanisme d'équilibrage interne au réseau n'a été implémenté.

f) *Have a mechanism that replicates the pieces of data across the nodes so that multiple nodes have it* : **OK**. Il s'agit des modes de stockage distributed, critical et hybrid.

g) *Extra* : Pas de fonctionnalité supplémentaire.

Question 4 : supervision

a) *Given an hostname, remotely spawn an agent on that node and have it join the topology* : **OK**. Il est possible d'ajouter un agent en utilisant la fonction `client:add_to_node/2`. Le second argument de cette fonction est le noeud d'accueil.

b) *Given an hostname, remove the corresponding node from the platform and return the hostname of a node still in the topology* : **OK**. De même, la fonction `client:kill_process/1` ne fait pas d'hypothèses sur le noeud sur lequel se trouve l'agent à tuer.

c) *Send data to a platform identified by the hostname of a node in the platform ;* : **OK**. La gestion des différents noeuds est en réalité parfaitement transparente

d) *Given an UUID, retrieve the corresponding piece of data from a platform identified by the hostname of a node in the platform* : **OK**

e) *Given an UUID, release the corresponding piece of data from a platform identified by the hostname of a node in the platform* : **OK**

f) *Query the topology for statistics ;* : **NOT OK**.

g) *Build a script that, given a set of nodes, automatically deploy the platform.* : **NOT OK**. Notre script de déploiement ne sert qu'à déployer une plateforme sur un seul noeud. Pour ajouter d'autres noeuds, il faut le faire manuellement.