Ali
Distribution de Tâches sur un Réseau

Guide Utilisateur

Thomas Bucher

Août 2004

Introduction

Quoi de plus rapide qu'un ordinateur ? Deux ordinateurs ! Développez récursivement cette formule n fois, et vous conviendrez qu'une ferme d'ordinateurs peut être largement plus rapide qu'un seul et même ordinateur.

Cette idée devient de plus en plus exploitée ces derniers temps, tant pour des raisons de coûts, que pour des raisons de performances... en effet, se doter d'une ferme de machines revient bien moins cher qu'un super-calculateur, et peut fournir des performances très comparables.

Une forme d'utilisation d'une ferme est la parallèlisation des traitements ; un gros calcul est divisé et exécuté parallèlement sur chaque machine de la ferme, et une fois les résultats de chaque calcul obtenus, ils sont regroupés ne formant qu'un seul et même résultat, comme si une seule machine avait exécuté ce calcul. Une autre utilité est la distribution de tâches ; le but est de décharger un serveur qui doit exécuter plusieurs tâches différentes à un moment donné... ces tâches seront alors redistribuées vers les machines constituant la ferme, et pourront donc être exécutées plus rapidement, individuellement, sur chaque machine.

Ali est un environnement permettant essentiellement d'effectuer de la distribution de tâches, soumises par l'utilisateur sous forme de fichiers de requêtes. Chaque tâche peut éventuellement être divisée en sous-tâches, qui pourront alors être traitées en parallèle, selon leurs inter-dépendances. Si les traitements sont relativement longs (par rapport, par exemple, au volume des données entrant en jeu) et la machine de l'utilisateur moins rapide que l'ensemble de la ferme, un gain de temps non négligeable peut être réalisé (sans compter la décharge de la machine de l'utilisateur)

1 Description de l'environnement Ali.

1.1 Description générale.

Ali se décompose en 4 entités différentes ; un serveur (Sali) qui répond aux requêtes des utilisateurs, des exécuteurs (Wali) qui exécutent les tâches envoyées par Sali, et un moniteur (Mali) qui gère une table indiquant l'état de chaque Wali. Enfin, la pièce manquante est l'utilisateur : c'est lui qui interagit avec Ali.

1.2 Le serveur (Sali).

Sali peut être unique, ou lancé sur plusieurs machines à la fois... l'utilité d'avoir plusieurs occurences de Sali sera expliquée ultérieurement. Sali est un serveur qui écoute sur un port donné, fixé par défaut ou configuré. Afin de pouvoir lui soumettre des tâches, il est nécessaire de connaître son nom (nom de sa machine hôte), et son port d'écoute. La soumission d'une requête se fera alors par le biais de la commande :

```
$submitter -n nom_du_serveur -p port_ecoute fichier_requete
```

Si le serveur est actif, s'il n'est pas surchargé et si vous êtes un client autorisé (i.e. présent dans le fichier d'autorisation du serveur), il répond à votre requête. Tout d'abord, il réceptionne le fichier de requête ; il l'analyse ensuite (notifie l'utilisateur si des erreurs sont trouvées dedans) et demande à recevoir les fichiers en entrée. Ensuite, pour chaque sous-tâche, il entre en communication avec Mali, lui demande un Wali adéquat pour la tâche à exécuter, et lui envoie cette tâche. Une fois que Wali a fini, il réceptionne les résultats, les stocke et tente de lancer d'autres sous-tâches. Enfin, une fois toutes les sous-tâches exécutées avec succès, les résultats sont renvoyés à l'utilisateur.

Il faut comprendre que pour chaque fichier d'entrée et de sortie, il y a transfert du client au serveur... c'est pourquoi avoir un serveur installé sur la machine même du client peut être bénéfique en terme de temps de transferts ; c'est pourquoi il peut y avoir plusieurs serveur actifs dans un environnement Ali.

1.3 L'exécuteur (Wali).

Wali est un démon tournant sur chaque machine de la ferme. Il répond aux demandes d'information provenant de Mali, et exécute les ordres de Sali. Wali dispose d'une liste de ressources disponibles (fichiers, applications ou

autres) qui sont communiquées au moniteur lorsqu'il les demande. Une tâche peut nécessiter une de ces ressources, et c'est ainsi que le moniteur décide lequel des exécuteurs est approprié pour ceelle-ci, en prenant aussi en compte la charge de la machine où tourne Wali. Ainsi, le rôle de Wali est simplement de lancer les tâches données par Sali, et de lui renvoyer les résultats.

1.4 Le moniteur (Mali).

Comme décrit plus haut, Mali possède des informations sur chaque exécuteur. À une fréquence donnée, le moniteur questionne chaque exécuteur sur son activité : sa charge CPU, ses ressources disponibles, etc... De plus, Mali attribue un score à la connexion réseau entre Wali et lui. Ces informations vont donc permettre à Mali de faire de la balance de charge, en recommandant tel ou tel Wali au serveur.

2 Les Tâches.

2.1 Description.

Une tâche est essentiellement composée de 3 éléments : les données en entrée, le traitement de ces données, et les données en sortie (le résultat du traitement). Cette base suffit pour réaliser des tâches relativement simples, telles que la compression d'un fichier, la concaténation de deux fichiers, etc. Ensuite, il est possible de rajouter de la complexité dans une tâche : on peut la décomposer en sous-tâches. Celles-ci, par exemple, pourront alors être dépendantes les unes des autres, i.e. exécutables uniquement lorsque les précédentes ont été réalisées. Ces dépendances peuvent réaliser des graphes plus ou moins complexes, mais pour un souci de clarté, il vaut mieux se limiter à des dépendances simples :

– "verticales" (ex. : 4 dépend de 3 qui dépend de 2 qui dépend de 1), ou
– "horizontales" (ex. : 5 dépend de 1, 2, 3 et 4).

Si l'exécution d'une sous-tâche échoue, ou que son délai est expiré, alors elle est relancée sur un autre Wali... ceci un certain nombre de fois. Si la tâche échoue continuellement, le serveur renvoit une erreur.

2.2 Le fichier de requête.

Le fichier de requête décrit la tâche à exécuter (divisée en une ou plusieurs sous-tâches) et qui sera envoyé (en plus des fichiers d'entrée) au serveur. Son écriture suit un langage relativement strict, composé de balises et de valeurs associées. Le fichier est alors structuré de la manière suivante :

- une entête générale (identifiée par la balise %HEADER)
- un ou plusieurs descriptifs de sous-tâche (identifié(s) par la balise %ID)

Chaque élement(entête ou descriptif) doit former un bloc compact (i.e. sans interligne ni commentaire). Des commentaires peuvent être ajoutés entre chaque bloc, ainsi qu'en début et fin de fichier. Ils sont identifiés par une ligne commençant par le caractère '#').

Attention : la casse des balises doit etre respectée !

2.3 Description des balises.

Avant de débuter la description des balises, voici d'abord un exemple de requête simple :

```
# ------------------------------------------------------------------------------ #
# exemple.req - Fichier de requete exemple.
# Ce fichier de requete donne la description d'une tache qui compresse deux
# fichier (fich1 et fich2) et les archive par la suite dans fich.tar.
# Les deux compressions sont independante (ce qui est logique), alors que
# l'archivage est dependant des deux precedentes taches.

%HEADER Requete exemple
%FIFILES fich1 fich2
%FOFILES fich.tar
%SEPARATOR ' '

# Compression du premier fichier
%ID 1
%CMD gzip fich1
%IFILES fich1
%OFILES fich1.gz
%LFILES
%REQUIRE gzip
%DEPS
%COST 1
%TIMEOUT 10
%RETRIES 0

# Compression du second fichier
%ID 2
%CMD gzip fich2.zip
%IFILES fich2
%OFILES fich2.gz
%LFILES
%REQUIRE gzip
%DEPS
%COST 1
%TIMEOUT 10
%RETRIES 0

# Archivage des deux fichiers compresses
%ID 3
%CMD tar -cf fich.tar fich1.gz fich2.gz
%IFILES fich1.gz fich2.gz
%OFILES fich.tar
%LFILES
%REQUIRE tar
%DEPS 1 2
%COST 2
%TIMEOUT 5
%RETRIES 0

# Fin du fichier de requete.
# ------------------------------------------------------------------------------ #
```

Entrons maintenant dans le vif du sujet avec le détail de chaque balise. Certaines balises sont obligatoires pour éviter toute erreur. Celles qui sont marquées d'une étoile (*) peuvent être omises, mais il est préférable de les mettre avec une valeur nulle pour expliciter l'absence de paramètre.

Le paragraphe d'entête est obligatoire afin de savoir quels fichiers doivent être envoyés au serveur, est quels fichiers seront renvoyés par celui-ci.

%HEADER [description]

Balise permettant d'identifier le paragraphe d'entête, qui rappelons le, ne doit former qu'un bloc, sans interlignes ni commentaires. Cette balise doit obligatoirement être la première dans le fichier. La description permet d'identifier des types de travaux envoyés au serveur ; on pourra par exemple mettre "Composition RGB" comme description pour identifier un ensemble de tâches qui consiste à réaliser une composition RGB. Cette description sera alors directement exploitée pour réaliser les statistiques au niveau du serveur. Si aucune description n'est donnée, la requête envoyée aura par défaut une description "Unknown".

%FIFILES [liste de fichiers]

Fichiers d'entrée primaires (First Input Files) ; ce sont ces fichiers que le client envoie au serveur lors de la soumission de la requête. Ce sont ces fichiers qui seront nécessaires au bon déroulement des tâches, tout du moins des premières... Dans notre exemple, fich1 et fich2 ne sont nécessaires qu'aux tâches 1 et 2, mais pas à la tâche 3, qui réutilise elle les résultats de 1 et 2.

%FOFILES [liste de fichiers]

Fichiers de sortie finaux (Final Output Files) ; ce sont les résultats finaux de la tâche globale ; il peuvent avoir été générés par n'importe quelle sous-tâche de la requête. Ces fichiers sont renvoyés par le serveur lorsque toutes les tâches sont finies (avec succès).

Les balises suivantes forment les paragraphes décrivant les différentes sous-tâches constituant la tâche globale. Au moins un paragraphe est obligatoire, puisque sinon la requête n'a plus lieu d'être.

%ID [entier]

Numéro identificateur de la tâche : il permet de nommer les tâches pour gérer les dépendances, et aussi de les identifier au sein du serveur. Ce numero doit être positif non nul, et il va de soit qu'il doit être unique pour chaque tâche.

Attention : cette balise doit obligatoirement se trouver en première position dans le paragraphe de tâche, car c'est cette balise qui l'identifie.

%CMD [chaine]

Commande à exécuter : c'est cette ligne de commande qui doit être exécutée, telle qu'on la lancerait depuis un shell. La ligne de commande doit impérativement être écrite en Bourne-Shell (pour éviter toute divergence de

syntaxe entre les différents shells). Petit rappel : pour rediriger à la fois la sortie d'erreur et la sortie standard vers le même fichier, il faut mettre la redirection `&> fichier`.

*%IFILES [liste de fichiers] **

Fichiers en entrée pour cette sous-tâche : ce sont les fichiers nécessaires à la bonne réalisation de la commande précédemment citée. Ces fichiers peuvent provenir des fichiers primaires, ou alors de résultats de tâches précedentes.

Valeur par défaut : liste vide

*%OFILES [liste de fichiers] **

Fichier en sortie pour cette sous-tâche : ce sont les fichiers générés par la commande, qui seront renvoyés au serveur et stockés pour renvoi au client ou utilisation par des tâches dépendantes.

Valeur par défaut : liste vide

*%LFILES [liste de fichiers] **

Fichiers liés dynamiquement : ce type de fichier permet au client de recevoir petit a petit les données, au fur et à mesure qu'elles sont générées par la commande. Le client doit alors prévoir l'existance de ce fichier dans son répertoire courant, fichier qui sera alimenté petit à petit par l'exécution de la commande. Cette fonction est très utile au sein d'une CGI, qui délocalise son traitement, mais doit en afficher l'évolution sur la page de l'utilisateur en "temps réel".

Valeur par défaut : liste vide

*%REQUIRE [liste d'items] **

Nom des ressources qui sont nécessaire à l'exécution de la tâche : on peut alors y mettre un nom d'application ou d'un fichier qui doit être présent sur la machine pour le bon déroulement de la tâche. Le nom de l'item doit correspondre parfaitement (casse comprise) à celui proposé par Wali pour que la tâche soit lancée dessus.

De plus, si les Wali sont bien configurés, la balise %REQUIRE peut servir pour lancer une tâche sur un Wali précis (identifié par un item qui correspond à son nom)... On peut facilement développer ce concept pour l'utilisation de groupes de Wali. Des exemples sont disponibles à la fin de ce document.

Valeur par défaut : liste vide

*%DEPS [liste d'entiers] **

Liste des tâches dont celle-ci est dépendante : cette sous-tâche attendra que toutes les tâches mentionnées dans cette balise soient finies pour être exécutée. La dépendance est surtout utilisée lorsque l'utilisation de fichiers intermédiaires est nécessaire, mais peut aussi servir à établir des points de synchronisation entre les différentes tâches.

Attention : toutes les tâches dont celle-ci dépend doivent être décrites en amont dans le fichier.

Valeur par défaut : liste vide

*%COST [entier] **

Coût prévu de la tâche : cette valeur permet au moniteur de prévoir la charge du Wali choisi après lancement de la tâche. Ceci permet de faire du load-balancing, même lorsque plusieurs tâches sont soumises en même temps au serveur.

On peut considérer que cette valeur varie de 1 à 100, 1 étant le coup d'une tâche non gourmande (par exemple 'ls' ou 'cat'), alors que 100 est le coup d'une tâche très gourmande (compression d'un film en divX, par exemple).

Valeur par défaut : 100

*%TIMEOUT [entier] **

Délais d'exécution de la tâche : au bout de ce délai (exprimé en secondes), la tâche est abandonnée sur l'exécuteur courant, et relancée ailleurs.

Valeur par défaut : 3600 (1h)

*%RETRIES [entier] **

Nombre de tentatives après un premier échec de la tâche. En effet, un échec peut être du à une erreur de l'utilisateur (omission d'un item requis par exemple), et la tâche peut bien fonctionner sur un autre Worker. Cependant, le relancement d'une tâche est un médiocre système de tolérance aux fautes, et dans un contexte de requêtes "rapides", il est préférable de laisser le nombre de re-tentatives à 0.

Valeur par défaut : 0

*%MPI [-1 | 0 | n] **

Sous-tâche exécutée en MPI : cette balise précise que la tâche doit être exécutée dans un environnement MPI ; elle n'est donc valable que pour les applications écrites en MPI (à charge pour l'utilisateur de s'informer sur cette technologie).

Lors de l'analye du fichier de requête, le serveur regroupe toutes les sous-tâches MPI en une unique tâche, qui lancera un environnement MPI avec en paramètre un schéma d'application comportant ce regroupement de sous-tâches. Différents modes de lancement de ces sous-tâches sont alors possibles :

-1 : la tâche sera lancée sur tous les processeurs disponibles (au sens de MPI)

0 : la tâche sera exécutée en local sur Wali

n : la tâche sera lancée sur n processeurs (au sens de MPI)

Les applications lancées par des tâches MPI doivent naturellement être écrites en MPI, sinon l'interêt du mode MPI est nul.

Un exemple typique de structure MPI est le traitement parallélisé d'une image. Prenons par exemple une image découpable en 1000 parties, chacune pouvant être traitée indépendamment. Admettons que nous avons un parc de 10 machines, dont une maître (qui sera alors le Wali chargé de la tâche). Un programme "découpeur" va alors découper l'image en 1000 parties, et envoyer le travail aux 10 "travailleurs", une partie pour chacun. Lorsque l'un d'eux fini, la partie d'image est envoyée au découpeur qui l'utilise pour reconstituer l'image résultante, et renvoit au travailleur une autre partie à traiter, et ainsi de suite...

L'exemple correspondant à cette illustration est disponible en fin de document. Brièvement, les paramètres des balises MPI correspondants aux tâches décrivant cette structure sont :

- 0 pour le "découpeur", qui doit être posté sur Wali pour pouvoir recevoir l'image de départ et renvoyer l'image résultat.
- 10 pour les 10 "travailleurs", qui seront lancés de manière transparente sur 10 machines du parc MPI.

Note : pour l'instant, le load-balancing n'est pas fonctionnel pour les applications tournant en MPI. De plus, l'utilisateur doit s'assurer que toutes les machines succeptibles de recevoir une tâche MPI sont aptes à l'exécuter.

3 Soumission d'une tâche

Comme décrit un peu plus haut dans le document, il est nécessaire de connaître le nom du serveur et son port d'écoute afin de pouvoir lui soumettre une tâche. Ces informations peuvent être obtenues auprès de votre administrateur.

Il existe deux manières possibles pour soumettre une tâche : l'utilisation d'une interface logicielle (le *submitter*), ou alors l'appel à une micro-API écrite en langage C (ANSI).

3.1 Le submitter

Le submitter est un programme écrit en C qui utilise la micro-API décrite plus bas. Il permet simplement d'envoyer la requête au serveur et de récupérer les résultats.

Différents modes de fonctionnement sont possibles, via les options passées sur la ligne de commande. L'utilisation la plus courante du submitter sera la suivante :

```
$submitter -n nom_serveur fichier_requete
```

Attention : les fichiers d'entrée doivent **obligatoirement** être présents dans le répertoire courant (celui d'où *submitter* est lancé). De plus, les fichiers de sortie ainsi que les fichiers dynamiquement liés, seront créés dans ce même répertoire. Si un fichier existe déjà alors qu'il doit être créé, le programme génèrera une erreur.

3.1.1 Description des options du submitter

`-n chaine` : le nom du serveur auquel la requête sera envoyée.
Valeur par défaut : la machine locale

`-p entier` : le port d'écoute du serveur.
Valeur par défaut : 6060 (valeur à l'heure actuelle)

`-q` : le submitter n'affichera rien sur sa sortie standard, hormis le fichier précisé par `-o` (le cas échéant). Cette option va justement de paire avec `-o` afin d'éviter toute perturbation de la sortie par des affichages intempestifs d'informations (tels que le temps d'exécution de la requête).
Valeur par défaut : false

`-o fichier` : `fichier` correspond à un fichier lié indiqué dans l'une des tâches de la requête. Au lieu de mettre à jour un fichier sur le disque, le submitter redirige toutes les données provenant de ce fichier lié vers sa sortie standard.

Valeur par défaut : sortie standard des tâches du job

`-e fichier` : même utilisation que `-o`, à la différence près que le fichier lié apparaît sur la sortie d'erreur.

Valeur par défaut : sortie erreur des tâches du job

`-c fichier` : nom du fichier de configuration à charger.

Valeur par défaut : $HOME/ali/conf/cali.conf

`-d répertoire` : répertoire de travail du submitter ; le submitter se placera automatiquement dans ce réperoire s'il est indiqué. Si celui-ci n'existe pas, le submitter ne tente pas de le créer et sort en erreur.

Valeur par défaut : .

`-d chaîne` : identificateur de la requête ; lorsque le submitter n'est pas muet, cette chaîne permet d'identifier quelle tâche s'est réalisée lorsque plusieurs sont lancées en parallèle.

Valeur par défaut : -

`-h` : affiche une aide succinte sur le *submitter*.

3.1.2 Fichier de configuration du submitter

En plus des options en ligne de commande, le submitter lit aussi sa configuration dans un fichier. Ce fichier est par défaut *$HOME/.ali/cali.conf*.

Les différentes options du fichier correspondent à celle en ligne de commande, mais possèdent une syntaxe différente :

`sali_name=chaine` : équivalent de l'option `-n`.

`sali_port=entier` : équivalent de l'option `-p`.

`quiet=true/false` : équivalent de l'option `-q`.

`stdout_file=fichier` : équivalent de l'option `-o`.

`stderr_file=fichier` : équivalent de l'option `-e`.

`id=chaîne` : équivalent de l'option `-i`.

`work_dir=répertoire` : équivalent de l'option `-d`.

3.1.3 Exemple utile d'utilisation des redirections

Visualiser la sortie des tâches que l'on souhaite exécuter n'est pas chose évidente, à priori. L'utilisation des deux options `-o` et `-e` couplée à une bonne description des tâches permet de combler cette lacune.

Voici l'exemple d'une tâche dont on veut récupérer non seulement le fichier de sortie, mais aussi la sortie standard et la sortie d'erreur :

```
[...]
%ID 1
%CMD gzip monfichier >fout 2>ferr
%IFILES monfichier
%OFILES monfichier.zip
%LFILES fout ferr
[...]
```

L'appel à submitter permettant de récupérer à la fois le fichier compressé, mais aussi les différentes sorties (standard et erreur), se fera donc de cette manière :

```
$submitter -o fout -e ferr requete_compression
```

Non seulement l'utilisateur trouvera *mon_fichier.zip* dans son répertoire courant, mais les différentes informations s'afficheront aussi sur son écran (selon la verbosité de gzip, dans notre exemple).

3.2 La micro-API

La micro-API se charge de réaliser la même chose que le submitter, mais a l'avantage de pouvoir être directement appelée dans le code d'une application (nécéssairement écrite en C/C++, pour l'instant).

Cette API se compose de deux fonctions :

- *submit_job_simple()* qui permet une soumission basique d'une requête, et
- *submit_job_complex()* qui permet une soumission plus paramétrable.

Ces deux fonctions sont bloquantes, jusqu'à ce que les résultats de la requête soient disponibles ou qu'une erreur intervienne. Elles renvoient :

- -1 si une erreur interne est arrivée,
- -2 si une erreur du coté serveur est arrivée,
- 0 si tout a bien fonctionné.

3.2.1 submit_job_simple

```
int submit_job_simple(char* host, char* reqbuf)
```

Le fonctionnement de cette fonction est relativement simple ; `host` est l'adresse du serveur (dans le format `nom:port`) et `reqbuf` le contenu de la requête.

Tout d'abord, la fonction analyse le contenu de la requête pour déterminer quels sont les fichiers à envoyer au serveur (les %IFILES et %OFILES dans le paragraphe $HEADER).

Ensuite, elle se connecte au serveur (d'adresse `host`) et entame le protocole de communication : demande d'exécution, envoi des fichiers, attente des résultats (ou de données de fichiers liés).

Enfin, lorsque le serveur renvoie les fichiers résultats, elle les écrit sur le disque, toujours dans le répertoire courant.

3.2.2 submit_job_complex

```
int submit_job_complex(char* host, char* reqbuf,
        char** ltargets, FILE** lstreams)
```

Cette fonction un peu plus complexe fonctionne de la même manière que la précédente, à part qu'il est possible de personnaliser la sortie des fichiers liés. Au lieu qu'ils soient écrits sur le disque, il est possible de les rediriger vers des flux donnés.

Les listes `ltargets` (noms des fichiers liés à rediriger) et `lstreams` (flux correspondants) doivent être toutes les deux finies par un pointeur nul. Chaque fichier lié indiqué dans `ltargets` se verra redirigé vers le flux correspondant donné dans `lstreams`. Un fichier lié ayant un flux égal à NULL sera traité comme par défaut : un fichier est créé sur le disque (avec comme nom celui donné dans la description de la tâche).

Voici le schéma simple d'utilisation de l'une de ces deux fonctions (la requête est supposée construite et l'adresse du serveur déterminée) :

```
[...]
exitval = submit_job_complex(serv_addr, reqbuf, ltargets, lstreams);
switch(exitval)
{
        case -1:
                printf("Une erreur interne est survenue.\n");

                /** code a effectuer en cas d'erreur...
                 * par exemple executer soit meme la tache
                 */
                break;
```

```
        case -2:
                printf("Une erreur est survenue sur Ali.\n");
                printf("Voici la pile des erreurs : %s", error_string(NULL));

                /** code a effectuer en cas d'erreur...
                 * par exemple executer soit meme la tache
                 */
                break;

        default:
                printf("La requete s'est bien deroulee! \n");

                /** code exploitant les résultats **/
                break;
}
[...]
```

Les fichiers liés présents dans les descriptions de tâches mais non indiqués dans la liste *ltargets* seront traités comme par défaut (i.e un fichier sera crée pour chacun d'eux).

4 Authentification - Sécurité

L'environnement Ali n'a pas la prétention d'être très sécurisé. En fait, pour pouvoir soumettre une tâche au serveur, il suffit de se trouver sur une machine autorisée. Les noms des machines autorisées par un serveur se trouvent dans le fichier *authorized_clients* du serveur. De la même manière, les exécuteurs possèdent aussi un fichier indiquant quels sont les serveurs (et moniteurs) autorisés ; de cette manière aucun serveur non fiable ne peut leur ordonner des tâches.

Il n'y a donc aucune gestion des noms d'utilisateurs, de leur droits, etc. C'est à l'administrateur d'autoriser ou non une machine, dont les utilisateurs doivent alors être fiables.

Cependant, les Wali ne tournent pas en root, et peuvent avoir des droits plus ou moins restreints sur leur machine. Ceci limite donc les risques, mais sans éviter le piratage "inter-tâche", par exemple.

5 Exemples modèles

Cette section a pour but de vous donner quelques exemples schématisant différents cas de requête possibles, et quelques utilisations spéciales. Il va de soi que les commandes utilisées dans les exemples ne sont qu'à titre illustratif. Par exemple, faire un *tar* via Ali ne présente pas beaucoup d'interêt en terme de performances.

5.1 Dépendance en colonne

Cet exemple vous montre comment créer une requête composée de plusieurs tâches séquentiellement dépendantes. Ici, on crée un fichier (résultat de la commande *ls*), on l'archive, puis on le compresse. Il n'y a pas de fichier en entrée, et le fichier de sortie est *sortie.tar.gz*.

```
%HEADER
%FIFILES
%FOFILES sortie.tar.gz

# Creation d'un fichier de donnees
%ID 1
%CMD ls > sortie.txt
%IFILES
%OFILES sortie.txt
%DEPS

# Archivage du fichier de donnees
%ID 2
%CMD tar -cf sortie.txt
%IFILES sortie.txt
%OFILES sortie.tar
%DEPS 1

# Compression du resultat
%ID 3
%CMD gzip sortie.tar
%IFILES sortie.tar
%OFILES sortie.tar.gz
%DEPS 2w
```

5.2 Dépendance en ligne

Dans cet exemple, les 3 premières tâches sont indépendantes, ce qui permet à Ali de les exécuter parallèlement. La dernière tâches travaille sur le résultat des 3 premières, c'est pourquoi elle est dépendantes de celles-ci.

```
%HEADER
%FIFILES fichier1 fichier2 fichier3
%FOFILES sortie.tar.gz

# Traitement fichier1
%ID 1
```

```
%CMD gzip fichier1
%IFILES fichier1
%OFILES fichier1.gz
%DEPS

# Traitement fichier2
%ID 2
%CMD gzip fichier2
%IFILES fichier2
%OFILES fichier2.gz
%DEPS

# Traitement fichier3
%ID 3
%CMD gzip fichier3
%IFILES fichier3
%OFILES fichier3.gz
%DEPS

# Utilisation des 3 fichiers
%ID 4
%CMD tar -cf fichiers.tar fichier1.gz fichier2.gz fichier3.gz
%IFILES fichier1.gz fichier2.gz fichier3.gz
%OFILES fichiers.tar
%DEPS 1 2 3
```

5.3 Utilisation des ressources de Wali

Comme décrit plus haute, la balise %REQUIRE permet de vérifier si un Wali propose bien certains items. Par exemple, le fichier */proc/1/cpu* (informations sur le processeur n°1) n'est pas disponible sur toutes les architectures. Pour lancer une tâche nécessitant ce fichier, il suffit de l'indiquer dans %REQUIRE :

```
%ID 1
%CMD cat /proc/1/cpu > cpu.txt
%IFILES
%OFILES cpu.txt
%REQUIRE /proc/1/cpu
%DEPS
```

5.4 Lancement de tâche sur Wali ciblé

La balise %REQUIRE permet non seulement de vérifier qu'une ressource (application, fichier) est présente, mais peut aussi servir à restreindre l'éxecution à un Wali précis ou un groupe de Wali. En effet, chaque Wali devrait proposer un item qui correspond à son nom, ce qui permet de lancer une tâche en précisant exactement sur qui la lancer. De la même manière, un Wali peut proposer un item correspondant à un groupe de Wali, ce qui permet de ne lancer la tâche que sur une des machines du groupe.

Ici, nous lançons une tâche sur un Wali précis (*cocat1.u-strasbg.fr*), et une autre sur un des Wali appartenant au groupe *cocat*.u-strasbg.fr* ; il est bon

de rappeler que ceci ne fonctionne que si les Wali sont bien configurés (autrement, vous recevrez un message du genre *"No suitable worker available"*).

```
%ID 1
%CMD date > data.txt
%IFILES
%OFILES date.txt
%REQUIRE cocat1.u-strasbg.fr

%ID 2
%CMD echo "petit test" > /tmp/test
%IFILES
%OFILES
%REQUIRE cocat*.u-strasbg.fr
```

5.5 Tâches MPI

Ceci est l'exemple qui reprend l'illustration donnée dans l'explication de la balise %MPI. Un *découpeur* est lancé sur une machine maître et des *travailleurs* sont lancés sur 10 machines (dont la machine maître peut faire partie).

```
%HEADER
%FIFILES image_originale
%FOFILES image_modifiee

%ID 1
%CMD decoupeur image_originale > image_modifiee
%IFILES image_originale
%OFILES image_modifiee
%MPI 0

%ID 2
%CMD travailleur
%IFILES
%OFILES
%MPI 10
```

6 Erreur courantes

L'utilisation du *submitter* ou de la micro-API peut engendrer des erreurs. Cette section vous présente quelques-unes des erreurs que vous risquez de rencontrer, leurs causes et comment y remédier.

Généralement, lorqu'une erreur arrive, le programme s'arrête et affiche la pile d'erreurs, de la plus ancienne à la plus récente. Chaque message d'erreur est précédé par le nom de la fonction où celle-ci s'est produite. Les messages d'erreur sont généralement assez clair pour que vous puissiez comprendre vous-même la cause, et donc y remédier.

```
Error@get_message: cannot create the file 'mesfichiers.tar' : File exists.
Error@submit_job_complex: cannot receive a message from Sali.
```

Typiquement, cette erreur survient lorsque vous appelez deux fois à la suite le *submitter*. En effet, vos fichiers de sortie sont déjà créés, et lorsque le *submitter* essaiera de les créer une seconde fois, il échouera.

Deux solutions sont possibles : soit vous effacez/déplacez les résultats après chaque exécution du *submitter*, soit vous vous placez dans un répertoire différent à chaque exécution.

```
Error@parse_req_buf: request contains no header.
Error@what_io_files: parsing the request file.
Error@submit_job_complex: getting information for i/o files.
```

Votre fichier de requête est mal formé. Respectez rigoureusement la syntaxe décrite dans la section 2 (page 4), et votre problème devrait disparaître.

```
Error@get_message: error sent from second peer.
Second peer Error Stack Trace:
Error@parse_req_buf: no CMD was specified (Task #1).
Error@client_handler: parsing the request buffer.
```

Même remarque que précédemment. À noter que dans ce cas, c'est le serveur qui remarque l'erreur est vous renvoie aussi sa pile d'erreur.

```
Error@submit_job_complex: cannot connect to the job-server (130.79.128.1:5050) :
Connection refused.
```

Vous avez donné de mauvais paramètres de connexion. Soit il n'y a pas de Sali sur la machine que vous avez indiqué, soit elle n'écoute pas sur le port donné. Voyez avec votre administrateur pour obtenir ces informations.

```
An error message was sent from the other side :
Error@get_suitable_worker: no worker is suitable for Task #1.
Error@launch_tasks: cannot choose a suitable worker for Task #1.
Error@client_handler: cannot launch tasks.
```

Le serveur vous indique qu'il n'a pas pu lancer les tâches nécessaires, car aucun Wali n'est propice au lancement de la tâche n°1 (dans cet exemple). Soit vous avez indiqué un item qu'aucun Wali ne possède, soit le Wali (ou les Wali) le possédant n'est pas disponible actuellement.

```
Warning@full_recv: no more data to receive
Error@get_message: cannot read  the message's code (nbread=0).
Error@submit_job_complex: cannot receive a message from Sali.
```

Ce genre d'erreur est critique : il y a eu un dysfonctionnement lors de la communication avec le serveur : une perturbation sur le réseau, une panne ou une erreur interne du serveur, etc. La seule solution est d'essayer plus tard ou de contacter l'administrateur si le problème persiste.

```
An error message was sent from the other side :
Task #1 fails. All its retries have been done.
```

Le serveur vous indique que la tâche n°1 ne cesse d'échouer, et a épuisé son "stock" de re-tentatives. Plusieurs causes à ces échecs sont possibles : vous utilisez un item spécifique, mais vous ne l'avez pas précisé dans la description de la tâche ; ou alors la commande que vous appelez est vouée à l'échec dans un certain contexte (contenu des fichiers, par exemple) ; ou bien vous avez tout simplement mal formé votre commande.

Dans tous les cas, il vous est possible de récupérer les sorties standard et erreur de la sous-tâche défectueuse, comme décrit dans la section 3.1.3 (page 12), et vous pouvez les analyser pour trouver la cause de l'erreur.

```
Error@get_message: error sent from second peer. The message is :
You are not an authorized client !.
Error@submit_job_complex: the Server cannot accept my request.
```

Vous n'êtes pas un client autorisé, i.e. le nom de votre machine n'est pas entré dans le fichier *authorized_clients* du serveur. Pour cela, il vous suffit de contacter votre administrateur pour qu'il le rajoute.

```
An error message was sent from the other side :
FOFILE 'fichier_test' doesn't exist !
```

Vous avez indiqué dans %FOFILES un fichier que le serveur ne peut vous renvoyer, car il n'existe pas. Celui-ci n'a peut-être pas été crée par l'une de vos tâches, ou alors vous avez peut-être fait une faute de frappe dans les noms. Vérifiez votre fichier de requête.

```
An error message was sent from the other side :
Error@launch_tasks: IFILE 'test' doesn't exist (Task #1).
Error@client_handler: cannot launch tasks.
```

Lors de l'envoi des ordres aux exécuteurs, le serveur n'a pas pu trouver le fichier *test*. Vous n'avez peut-être pas indiqué ce fichier dans %FIFILES, ou alors celui-ci n'est pas créé comme fichier intermédiaire. Vérifiez votre fichier de requête.

Conclusion

La distribution de tâches sur un réseau est un concept largement utilisé pour effectuer de gros calculs. Ali vous permet de lancer de manière simple certaines applications sur une ferme de Wali, avec plus ou moins de souplesse. Ce n'est qu'une solution supplémentaire parmi toutes celles disponibles actuellement, avec ses avantages et ses inconvénients. Toutes remarques sont les bienvenues pour permettre d'améliorer et de faire évoluer ce projet.