LibGridRPCDM, une surcouche intéropérable et un gestionnaire de données pour le standard GridRPC

Hadrien Croubois

27/05/2013 - 16/08/2013

Sous la direction de M. Yves Caniou,
Maître de conférences à l'Université de Lyon
Délégation CNRS au JFLI, Université de Tokyo

Remerciements

À l'issue de mon stage, je tiens à remercier tout particulièrement :

M. Yves Caniou,
pour m'avoir donné l'opportunité de réaliser ce travail ainsi que pour sa disponibilité et ses conseils tout au long de mon stage ;

M. Philippe Codognet, M. David Nowak, M. Alejandro Arbelaez,
pour leur accueil chaleureux ;

M. Gaël Le Mahec,
pour l'aide qu'il m'a apportée durant l'acquisition du code et pour m'avoir éclairé lors de difficultés avec DAGDA ;

M. Hidemoto Nakada,
pour sa contribution à l'intégration de Ninf ;

M. Eddy Caron,
pour m'avoir orienté vers le JFLI lors de ma recherche de stage ainsi que pour l'aide qu'il m'a apportée vis-à-vis de DIET et iRods ;

L'université de Tokyo.

Table des matières

Partie 1

Introduction

1.1 Contexte et vocabulaire

Le terme *grid computing* naît au début des années 1990. Avec l'augmentation des ressources informatiques, on rêve alors de pouvoir mettre toute cette puissance en commun, comme c'est le cas sur le réseau électrique (Power-Grid en anglais). Il a cependant fallu attendre 1999 et l'apparition de SETI@Home [1] pour voir le concept de *desktop computing* atteindre les postes de travail des particuliers.

Au delà des besoins en puissance de calcul, on utilise aujourd'hui des applications grid ou cloud pour la gestion de grandes quantités de données. La réplication des données sur plusieurs nœuds du réseau permet ainsi d'assurer la pérennité des données en cas de crash d'une des machines tout en assurant un accès rapide aux données.

La distribution des calculs se fait généralement en passant par un *grid middleware* (aussi appelé intergiciel en français).

Les middleware devant se charger de la distribution des calculs sur la grille, il est également nécessaire de gérer l'ensemble des données entrantes et sortantes. À ce titre, chaque middleware est généralement accompagné d'un *data manager* (ou gestionnaire de données) qui se charge du transfert et de la réplication des données.

1.2 Le standard OGF GridRPC et GridRPC Data Management

Avec l'objectif d'assurer l'interopérabilité, au moins comportementale, des grid middleware, l'OGF a standardisé en 2007 l'API GridRPC[6] (Grid Remote Procedure Call) reposant sur le paradigme du même nom.

Ce paradigme (voir figure 1) repose sur la notion d'agent qui enregistre les services mis à disposition par les serveurs de calculs (1). Quand un client veut soumettre une tâche de calcul, il n'a donc qu'à interroger l'agent (2) qui lui retourne un *handle* (3).

Une fois le *handle* en possession du client, ce dernier n'a alors qu'à appeler (4), de manière synchrone ou asynchrone, le serveur décrit par le handle qui lui renverra directement les résultats (5).

L'une des forces de ce modèle est que, au moment de la création du *handle*, l'agent peut choisir sur quelle unité de calcul effectuer les opérations décrites par le client afin de distribuer correctement la charge sur le réseau.

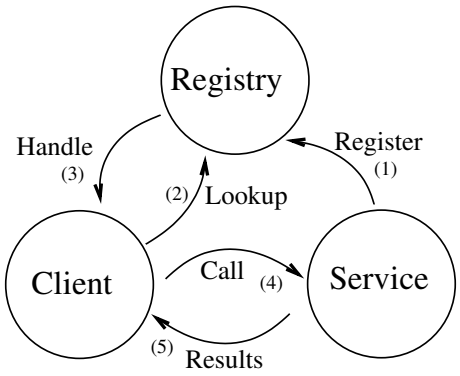

Figure 1 – Le paradigme GridRPC

Ce paradigme permet ainsi de simplifier les appels côté client tout en laissant une grande liberté aux administrateurs des plates-formes de calcul, notamment en terme d'ordonnancement et de gestion des contrats.

En juin 2011, l'OGF a étendu l'API GridRPC par l'ajout d'une interface de data management.

L'API GridRPC DataManagement[8] décrit notamment un format de données unifié ainsi que les fonctions de gestion associées.

1.3 Objectifs de la bibliothèque libGridRPCDM

L'objectif de la bibliothèque libGridRPCDM est double :

- Fournir une implémentation du standard GridRPC Data Management suivant les recommandations de l'OGF ;
- Proposer une interface GridRPC unifiée permettant d'appeler des services présents sur plusieurs middleware de manière transparente.

1. SETI@Home – `http://setiathome.berkeley.edu/`

Partie 2

Acquisition du code

2.1 Architecture

Après un premier prototype, utilisé comme démonstrateur lors de la standardisation de l'API GridRPC Data Management en 2011, la bibliothèque a été reprise en profondeur en mars 2013. Cette nouvelle version posait les bases d'un outil plus complet et modulaire. Son état ne permettait cependant pas une utilisation réelle.

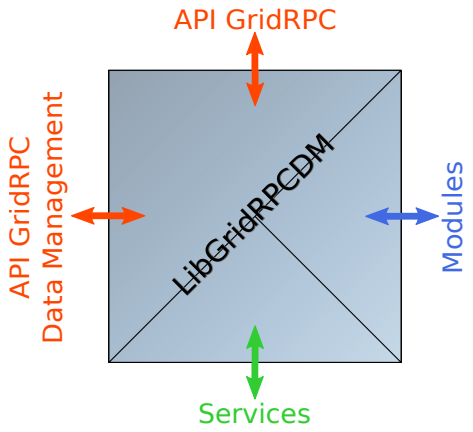

FIGURE 2 – Les différentes interfaces de la bibliothèque

L'objectif primaire de mon stage étant de compléter cette bibliothèque afin de la rendre utilisable, il m'a d'abord fallu étudier le travail déjà accompli. Cette tâche à été rendue complexe par l'absence de documentation. Cela m'a pris une semaine pour bien assimiler les différents éléments alors présents :

- La classe `grpc::GRPCMgr` (depuis renommée en `grpc::Core`) qui gère notamment la procédure d'initialisation et les mapping mémoires ;
- Les relations d'héritage entre les différents types de modules ;
- La procédure de chargement des modules ;
- Les méthodes de gestion des méta données ;
- Des outils complémentaire, aussi bien internes (gestion de certaines structures de données) qu'externes (appel à `CMake`).

2.2 Initialisation et chargement des modules

À cause de la nature de la bibliothèque, la procédure d'initialisation a un rôle central, qu'il s'agisse du chargement des modules ou de l'initialisation des différents singletons.

Cette procédure est lancée par le biais de la fonction `grpc_initialize()` mise à disposition par la bibliothèque. Un fichier de configuration au format `cfg`, passé à cette fonction, permet de régler les différents paramètres ainsi que de lister les modules à charger (voir table 1).

TABLE 1 Listes des options réglables dans le fichier de configuration

Option	Type (valeur par défaut)	Description
default_middleware	string	Middleware à utiliser par défaut
memory_protocol	string	Protocole à utiliser pour le transfert de données locales
file_protocol	string	Protocole à utiliser pour le transfert de fichiers
IO_try_count	int(3)	Nombre maximum de tentatives pour les requêtes get/put
max_simul_download	int(10)	Nombre maximum de téléchargements simultanés de données
max_simul_upload	int(10)	Nombre maximum d'upload simultanés de données
verbose	bool(false)	Active/Désactive l'affichage d'informations complémentaires

Le fichier de configuration étant construit comme un arbre, l'ajout d'un module consiste à ajouter un nœud comportant le chemin vers le fichier binaire. Lors du chargement du module, le type de ce dernier sera automatiquement détecté. Pour les modules de type *data management* et *middleware*, il est possible d'ajouter des fichiers de configuration spécifiques qui seront fournis aux fonctions d'initialisation respectives (voir code 1).

Lors du chargement d'un module, un objet est instancié, contenant les pointeurs vers les fonctions présentes dans les fichiers binaires. Les objets créés héritent tous d'une même classe `grpc::Module` qui intègre le nom et le type des modules simplifiant ainsi la gestion des objets modules.

Les adresses des modules créés sont ensuite indexées dans différentes structures qui permettent d'y accéder rapidement, que ce soit par leur nom ou par les protocoles pris en charge. Des informations complémentaires sur l'état des modules (réussite de l'initialisation notamment) sont aussi conservées.

La figure 11 (en annexe) présente les différentes étapes du processus de chargement des modules.

CODE 1 Liste de modules à charger telle qu'attendue par le fichier de configuration

```
[webDav]
  module = ../install/share/modules/libgrpc-webdav.so
  datamanager_config = webdav.cfg
[dagda]
  module = ../install/share/modules/libgrpc-dagda.so
[diet]
  module = ../install/share/modules/libgrpc-diet.so
  middleware_config = dietCFG/client.cfg
```

La procédure d'initialisation a été quelque peu modifiée lors de ce stage du fait des améliorations apportées à la bibliothèque :

- L'ajout rapide d'une fonction d'initialisation pour les modules de *data management* a demandé l'ajout d'une seconde option pour passer des fichiers de configuration. Un module pouvant combiner plusieurs fonctionnalités (*middleware* et *data management*), il était important de pouvoir passer deux fichiers de configuration différents et donc d'avoir deux paramètres distincts ;
- La nécessité d'une fonction de suppression de données, imposée par le standard GridRPC Data Management, a également demandé une évolution de la structure des modules ;
- Les différents singletons utilisés comme gestionnaires sont également initialisés durant cette phase ;
- Enfin les paramètres `IO_try_count, max_simul_download` et `max_simul_upload` ont été ajoutés et servent à l'initialisation du singleton `grpc::IOManager` gérant les connexions entrantes et sortantes.

2.3 Compatibilité des headers

Afin de pouvoir interfacer la bibliothèque avec des applications utilisateur, les différentes fonctions des interfaces définies par les standards GridRPC et GridRPC Data Management sont déclarées dans un fichier .h spécifique. Les middleware *GridRPC-compliant* intégrant déjà une interface GridRPC, il est possible de se retrouver confronté à des conflits de nommage, les structures et fonctions propres au standard étant déjà déclarées par le middleware. Ce problème potentiel a néanmoins pu être résolu, malgré l'impossibilité d'utiliser des namespaces (inexistants en C), via un système de macro permettant de désactiver individuellement les différentes déclarations (voir figure 3).

Les structures décrites par l'API GridRPC étant déjà utilisées par le middleware *GridRPC compliant*, la définition des structures correspondantes par la bibliothèque peut être surchargée à l'aide d'une macro. Ainsi le type `grpc_sessionid_t` sera changé en `_grpc_sessionid_t` par la déclaration de `GRPC_SESSIONID_DECLARED`. Une telle déclaration, avant l'inclusion du header de la bibliothèque, permet de faire coexister les deux structures au sein du module. Afin de faciliter l'accès aux différentes structures, les noms, variables par l'emploi de ces macros, des structures de la bibliothèque sont accessibles car déclarés comme commande de préprocesseur.

Partie 3 Évolution générale du code et des outils

Avant de pouvoir implanter de nouvelles fonctionnalités à proprement parler, il m'a été nécessaire de développer un certain nombre d'outils qui se sont par la suite relevés extrêmement utiles. Afin de simplifier la tâche des développeurs de modules et de services, bon nombre de ces outils ont été mis à disposition et sont compilés au format C.

3.1 Refactorisation du code

Afin de clarifier la division du code et ainsi de simplifier l'assimilation du code par toute personne désirant contribuer au projet, les sources ont été refactorisées en profondeur. Il s'agit de faire la différence entre le code purement interne de la bibliothèque (C++), l'interface GridRPC / GridRPC Data Management et les fichiers d'en-tête partagés. Ce

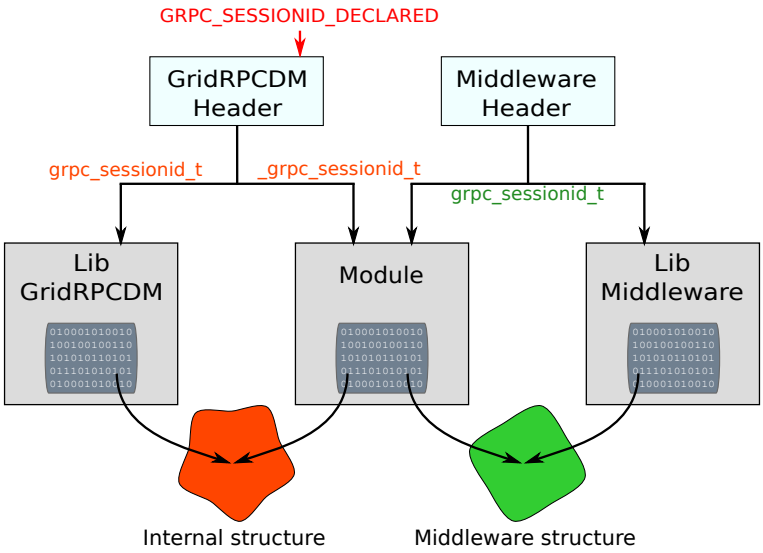

FIGURE 3 – Résolution des conflits de nommage par l'utilisation de macros

travail se révélera par la suite essentiel au moment d'intégrer de nouvelles fonctionnalités ainsi que des services ad-hoc pour les middleware GridRPC.

Le code se décompose selon les dossiers suivants :

- Un dossier `include/`, contenant les fichiers entête C (`.h`) décrivant les interfaces de la bibliothèque, qu'il s'agisse des API GridRPC, GridRPC Data Management, ou des services mis à disposition pour le développement de modules et de services. L'ensemble de ces fichiers est installé avec la bibliothèque pour permettre la compilation externe de modules compatibles ;
- Un dossier `src/`, contenant les fichiers C++ (`.cc`) constituant les différentes fonctionnalités de la bibliothèque ainsi que les entêtes associés (`.hh`). On y trouve aussi, bien séparés du reste de la bibliothèque, les fichiers `API_Extended.cc`, `API_GridRPC.cc` et `API_GridRPCDM.cc` contenant les sources des fonctions des différentes API. Ces fichiers sont compilés de manière à permettre la liaison des fonctions contenues avec du code C. L'ensemble de ces fichiers permet la compilation des binaires constituant la bibliothèque ;
- Un sous dossier `src/testing/`, contenant les sources des clients et des serveurs de la suite de tests ainsi que des scripts permettant de lancer automatiquement les différentes tests en passant par DIET ou par Ninf ;
- Un dossier `modules/` contenant les sources des différents modules. Ces modules sont compilés séparément et installés dans un dossier spécifique.

La refactorisation du code a notamment été l'occasion de corriger quelques erreurs de définitions des fonctions qui ne correspondaient pas exactement aux spécifications des documents de l'OGF.

Cette phase d'assimilation du code a aussi été l'occasion de créer un script bash `./generator.sh` permettant de construire et d'installer le projet de manière automatique. Ce script fait notamment appel à l'utilitaire de compilation `cmake` qui s'avère particulièrement utile pour la gestion des dépendances pour la compilation du projet. Au delà de la construction du projet, ce script permet d'activer la constuction de la suite de tests et se charge par la même occasion de générer des scripts qui simplifieront le lancement des différents tests.

3.2 Outils de gestion des données et des méta données

Les données sur lesquelles reposent l'API GridRPC Data Management (structure `grpc_data_t`) [2] étant décrites par une structure relativement complexe, plusieurs outils ont dû être développés pour leur gestion.

3.2.1 Sérialisation des données et des méta données

Les GridRPC middleware ne gérant pas la transmission de structures complexes, il fallait trouver un moyen de transmettre la structure `grpc_data_t` aux services middleware. Pour cela les `grpc_data_t` sont sérialisés et transmis par les middleware au service où une copie est alors créée en désérialisant la chaîne obtenue [3].

Il a donc fallu mettre en place des méthodes de sérialisation/désérialisation sur ces structures C (voir code 2).

2. Il s'agit en fait de méta données comprenant notamment le type et la taille de données décrites ainsi qu'une liste des URIs où cette donnée est présente et/ou doit être envoyée.
3. La création d'une telle copie peut présenter de nombreux problèmes. Voir section 7.1

Pour cela, on utilise une classe C++ (`grpc::GRPCData`) contenant les mêmes informations que la classe C, mais encapsulées dans des conteneurs de la bibliothèque standard C++ (les *null-terminated list* sont notamment converties en `std::list`). Cette classe est ensuite sérialisée à l'aide des outils mis à disposition par la bibliothèque *boost*.

CODE 2 Prototypes des outils de sérialisation proposés par l'API étendue

```
grpc_error_t grpc_data_serialize(grpc_data_t* data, char** str);
grpc_error_t grpc_data_deserialize(grpc_data_t* data, const char* str);
```

3.2.2 Extension de la gestion des URIs

Les structures de données décrites par le standard *GridRPC Data Management* utilisent non seulement des URIs[3] pour définir l'adresse, le protocole et le port où les données sont présentes, mais aussi d'autres informations telles que le mode de persistance des données à envoyer. Afin de simplifier la gestion de ces différentes données, j'ai mis en place une classe C++ `grpc::extendeduri` qui contient notamment l'URI et le mode mais aussi d'autres informations complémentaires telles que la présence d'un middleware sur le serveur décrit par l'URI. Cette dernière donnée sera particulièrement utile pour la distribution des transferts (voir Section 5.2).

Il est de plus important d'être capable de récupérer certains champs spécifiques présents dans les URIs. Pour cela la classe `grpc::extendeduri` intègre une sous classe `grpc::extendeduri::uriparser` permettant de parser les différentes informations contenues dans l'URI.

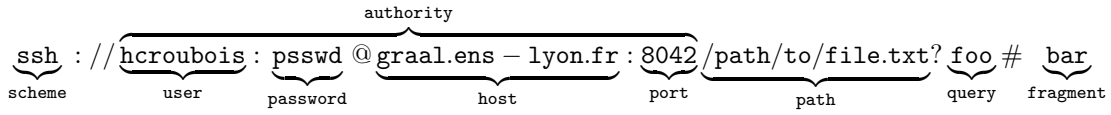

FIGURE 4 – Différents éléments renvoyés par la méthode de parsing des URIs

Pour des raisons de performance, un tel découpage n'est pas fait à la construction de l'objet `grpc::extendeduri` mais automatiquement lors de l'appel au champs `grpc::extendeduri::parsed()`. L'objet `grpc::extendeduri::uriparser` alors associé est gardé en mémoire afin d'éviter qu'il ne soit recalculé à chaque appel. Il sera automatiquement détruit avec l'objet `grpc::extendeduri` qui l'a produit.

3.3 Outils de gestion des requêtes

La bibliothèque doit également pouvoir gérer de nombreux types de requêtes, qu'il s'agisse de transferts de données, d'appels middleware ou autre. Il est donc important d'avoir les outils nécessaires à la soumission et au suivi de requêtes très variées.

Dans l'optique de construire un système à la fois simple d'utilisation et cohérent, j'ai fait le choix de mettre en place un framework proposant les mécanismes nécessaires à la gestion de requêtes asynchrones. Ces mécanismes doivent par la suite être utilisables pour l'attente et la gestion de différents types de requêtes, qu'il s'agisse de tâche de calcul ou de transfert de données.

3.3.1 Requêtes asynchrones

Dans sa version actuelle (2.0), le framework de requêtes est centré sur une classe abstraite `grpc::Request<T>` qui décrit l'interface entre les différentes requêtes et le reste de la bibliothèque.

Ce framework a beaucoup évolué pour répondre aux différents besoins rencontrés. La version 2.0 a ainsi vu l'apparition d'une seconde information retournée. En plus d'un code d'erreur (codé sous forme d'un `int`), les requêtes disposent d'un template pour permettre de varier le type de données associé à la requête associé et que retourne la requête. Il est ainsi possible d'associer une requête de type tâche de calcul avec le descripteur de session, ce qui permet d'avoir plus d'informations sur la requête ayant provoquée un déblocage dans le cas d'une attente bloquante.

Cette classe abstraite est implémentée dans la classe `grpc::RequestImplementation<T>` dont hérite par la suite la plupart des autres requêtes. Les mécanismes d'attente sont mis en place à l'aide d'un sémaphore utilisé comme *lock interprocess*. Ce lock est ainsi verrouillé à la création de la requête (par le thread initialisant la requête) et déverrouillé à la fin de l'exécution (potentiellement par un autre thread). L'attente de la requête, tout comme la simple vérification de son état, reviennent ainsi à de simples opérations sur le sémaphore et sont donc entièrement *thread safe*.

3.3.2 Requêtes de contrôle (`WAIT_ANY`)

Afin d'aider à la gestion des requêtes, la classe `grpc::MonitorRequest` permet de suivre l'évolution de plusieurs requêtes et d'attendre qu'un certain nombre d'entre elles finissent. En définissant le seuil d'attente à 1, on peut ainsi attendre la première requête terminée et implémenter les fonctions `grpc_wait_or`, `grpc_wait_any` et `grpc_data_wait`.

Ce modèle a pour but d'être le plus polyvalent possible et pourrait permettre de créer des DAGs de requêtes, ce qui laisse ainsi une grande marge d'évolution à la bibliothèque. Afin de limiter la consommation en ressources, cette classe n'utilise pas de thread dédié et repose sur un système complexe de référencement.

3.3.3 File de requêtes et threads d'exécution

Toujours dans l'optique de construire un système de gestion des requêtes, j'ai implémenté une file d'attente procédant à l'évaluation en parallèle d'un nombre limité de requêtes. C'est le rôle de la classe `ReqController`.

Cette classe hérite de manière privée de `std::list<grpc::Request<T>*>` ce qui en fait naturellement un conteneur de requêtes. Les fonctions d'ajout et de récupération permettent ainsi de stocker les requêtes en attente d'exécution. Ces fonctionnalités ont été rendues indépendantes du reste afin de simplifier toute évolution, comme par exemple le passage d'un comportement FIFO à une file de priorité qui permettrait de définir différents degrés de priorités aux requêtes.

Une seconde partie de la classe se charge d'exécuter parallèlement un certain nombre de requêtes. Cela est fait par l'utilisation combinée de threads exécutant une fonction membre dédiée et de sémaphores pour contrôler le nombre de threads actifs (voir code 3).

CODE 3 Fonctionnement de la classe `grpc::ReqController`

[Ajout]
- Ajout de la requête en queue de file
- Si limitation des exécutions parallèles
  - Tentative d'acquisition d'une ressource par le sémaphore
  - Si aucune ressource n'est disponible
    - return
- Lancement d'un thread d'exécution
- return

[Exécution]
- Tant qu'il y a des tâches en attente
  - Exécution de la tâche en tête
- Si limitation du nombre d'exécutions parallèles
  - Relâche d'une ressource par le sémaphore
- return (fin du thread courant)

L'idée derrière la limitation des requêtes parallèles est notamment d'éviter l'engorgement[5] en limitant le nombre de téléchargements simultanés (`IRequest`/`ORequest`). Cette limitation est potentiellement amenée à être ré-évaluée au cours de l'exécution du programme (de manière à suivre l'évolution des performances réseau). En prévision, une méthode de modification du nombre maximum de requêtes parallèles a donc été mise en place. Elle se charge notamment de :

- Tuer les threads en trop à la fin de l'exécution de la requête en cours (cas où le nombre de ressources diminue) ;
- Créer de nouveaux threads s'il y a des requêtes en attente (cas où le nombre de ressources diminue) ;
- Mettre à jour le sémaphore de manière adéquate.

3.3.4 Classes héritant du framework

La classe `grcp::Request<T>` présente dans le framework de requêtes est utilisée dans les différentes requêtes de la bibliothèque. Ces requêtes, au constructeur privé, ne peuvent être construites que par les managers (voir table 2).

Le rôle de ces managers est de construire les requêtes et de les ajouter dans un `grpc::ReqController` qui se chargera de les exécuter en parallèle. Les managers se chargent de plus de référencer les requêtes, de les désallouer pour limiter l'impact mémoire et de mettre en place des mécanismes d'attente en fin de programme.

Toutes les requêtes héritant de la même classe, les mécanismes de contrôle implémentés par la classe `grpc::MonitorRequest` permettent notamment d'attendre la première tâche terminée parmi un ensemble de `IRequest`, de `ORequest`, de `IRequest` et de `Dispatcher`.

Table 2 Liste des classes héritant du type `grpc::Request<T>`

Classe	Manager	Type de résultat	Rôle
IRequest	IOManager	`grpc_data_t*`	Téléchargement de données (transfert entrant)
ORequest	IOManager	`grpc_data_t*`	Upload de données (transfert sortant)
IORequest	IOManager	`grpc_data_t*`	Transfert de données locales (téléchargement puis upload)
CallRequest	CallManager	`grpc_sessionid_t*`	Appel à un middleware
Dispatcher	TransferManager	`grpc_data_t*`	Distribution des tâches de transfert

On a de plus, et ce de manière naturelle, des méthodes pour procéder à des attentes sur les appels middleware vers différents middleware. Il est dès lors possible d'attendre en parallèle un appel vers un service DIET et vers un service Ninf.

3.4 Structure interne de la bibliothèque

Lors de ma prise en main de la bibliothèque, la classe `grpc::GRPCMgr` (depuis renommée `grpc::Core`) était la seule classe statique et constituait le cœur de la bibliothèque. Il m'a cependant été nécessaire de mettre en place d'autres structures statiques ce qui a profondément modifié l'organisation interne de la bibliothèque.

Afin de simplifier et de clarifier la mise en place de singletons, une classe C++ `grpc::Singleton<T>` a été spécifiquement développée. Cette classe permet de transformer n'importe quelle autre classe A en singleton en quelques étapes :

- Rendre le constructeur de la classe A privé pour éviter les constructions non voulues ;
- Rendre la classe A amie avec la classe `grpc::singleton<A>` ainsi d'autoriser la construction d'une instance de A par l'interface singleton ;
- Faire hériter (de manière publique) A de `grpc::singleton<A>` afin de mettre à disposition les différentes méthodes propres au singleton.

Les méthodes propres aux structures singleton sont décrient dans la table 3.

Table 3 Liste des méthodes propres aux classes singleton

Méthode	Description
`protected :`	
Singleton()	Mise en place de l'appel automatique de la méthode kill() à la fermeture du programme.
~Singleton()	
`public :`	
static void init(...)	Construction de l'instance unique à partir des arguments fournis.
static T& instance()	Accès à l'instance singleton.
`private :`	
void kill()	Destruction du singleton (appelée automatiquement à la fermeture du programme).

Partie 4 Évolution des modules

4.1 Évolution de la structure des modules

Avec l'écriture de nouveaux modules de communications, il a été nécessaire de faire évoluer la structure des modules pour répondre à de nouveaux besoins.

4.1.1 Initialisation des modules de data management

Certaines bibliothèques, telles que la bibliothèque *Neon*[13] qui gère les communications `webDav`, nécessitant une initialisation statique, il a été nécessaire d'ajouter une fonction d'initialisation aux modules de *data management*. Une telle fonction, jusqu'alors réservée aux modules *middleware*, a nécessité de nombreuses adaptations :

- Ajout d'options pour le passage d'un fichier de configuration permettant de paramétrer les modules de *data management* ;
- Ajout d'un tableau indiquant l'état d'initialisation des modules de *data management* ;
- Modification de la procédure d'allocation et d'initialisation des modules ;
- Modification de la procédure d'identification des protocoles supportés.

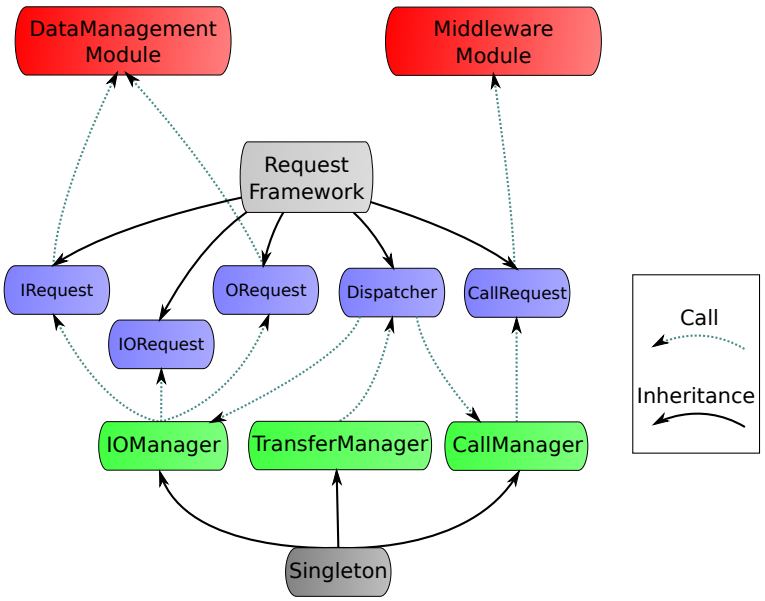

FIGURE 5 – Diagramme des dépendances entre classes

On assure ainsi la plus grande compatibilité possible avec les bibliothèques utilisées par les modules tout en assurant la cohérence des informations permettant le fonctionnement de la bibliothèque. Il est en effet important, dans le cas où deux modules géreraient un même protocole et où l'un des deux modules ne serait pas correctement initialisé, que le module appelé soit bien le module fonctionnel.

4.1.2 Fonction de suppression des données

La fonction `grpc_data_free`, telle que décrite dans l'API GridRPC Data Management, demandant la suppression des données décrites, il est nécessaire que les modules de *data managemment* soient capable de procéder à une telle opération. Cette fonctionnalité n'ayant pas été prévue par le prototype dont est issu la bibliothèque, cette fonctionnalité a dû être ajoutée aux modules de *data management* afin de permettre la mise en place du comportement décrit par le standard.

4.2 Évolution des modules préexistants

Lors de la remise à zéro de la bibliothèque en mars 2013, les premiers modules mis en place ont été les modules DIET et DAGDA. Au delà de la compatibilité avec DIET, ces deux modules devaient permettre de tester toutes les fonctionnalités mises en place au fur et à mesure. Certains problèmes m'ont cependant amené à modifier partiellement ces deux modules.

4.2.1 DIET

Le module DIET semblant dater du prototype de bibliothèque développé en 2011, il a été nécessaire de modifier les appels pour utiliser les nouveaux outils de sérialisation des données.

En voulant gérer les appels asynchrones à l'aide des outils proposés au niveau middleware, plusieurs problèmes sont apparus. Il semble en effet que la fonction `diet_async_call` provoque des erreurs quand l'environnement dans lequel elle est lancée est fermé trop rapidement (problème de durée de vie de certains objets). Ce problème est apparemment connu et il m'a été conseillé de mettre en place les appels asynchrones à l'aide de threads, l'asynchronisme étant géré par la bibliothèque. C'est ce qui est aujourd'hui fait à l'aide du framework de requêtes (voir Section 3.3).

4.2.2 DAGDA

Plusieurs soucis se sont manifestés lors de l'utilisation du module DAGDA préexistant. Il a notamment été nécessaire de réécrire les fonctions de transferts pour mieux prendre en compte les différents types proposés par le standard *GridRPC Data Management* ainsi que pour simplifier les méthodes de gestion des alias.

Plusieurs problèmes relatifs aux *alias* se posent encore. Il semble en effet être impossible de créer un alias depuis un SeD. À cela s'ajoute le fait qu'il ne semble pas possible de mettre à jour un alias en cas de modifications des données. Enfin, lors du téléchargement d'une donnée à partir d'un *dagdaID*, DAGDA retourne : `dagda_get_data: unknown data type`.

4.3 Nouveaux modules

4.3.1 Ssh *et* Rsync

Afin de permettre le stockage de données sur la plupart des machines, la gestion du protocole Scp a dès le début semblé indispensable. Le protocole Scp ne gérant que des fichiers, les outils de sérialisation jouent un rôle essentiel dans son fonctionnement. Le transfert de données par Scp se décompose donc en plusieurs étapes (voir code 4).

code 4 Protocole de gestion des données pour des data manager reposant sur des systèmes de fichiers

[Download]
- Création d'un fichier temporaire dans `/tmp`
- Copie depuis le serveur à l'aide de la commande `scp`
- Désérialization du fichier temporaire vers la mémoire vive (si type différent de `GRPC_FILE`)
- Suppression du fichier temporaire (si type différent de `GRPC_FILE`)

[Upload]
- Création d'un fichier temporaire dans `/tmp` (si type différent de `GRPC_FILE`)
- Sérialization des données dans le fichier temporaire (si type différent de `GRPC_FILE`)
- Copie vers le serveur à l'aide de la commande `scp`
- Suppression du fichier temporaire (si type différent de `GRPC_FILE`)

Ce modèle a par la suite été repris dans tous les modules de *data management* fonctionnant par fichiers (iRods, WebDav). Il apparaît donc que lors du transfert d'une donnée d'un *data manager* à un autre, si les deux *data manager* reposent sur des fichiers, l'utilisation du type `GRPC_FILE` permet d'éviter l'allocation de mémoire ainsi que certaines recopies.

Toujours dans l'optique de fournir de nombreuses possibilités aux utilisateurs, ce module se décline sous deux versions : une version Ssh utilisant la commande `scp` et une version Rsync utilisant la commande `rsync`. Une version utilisant la commande `oarscp` est également prévue pour permettre le fonctionnement de la bibliothèque sur *Grid5000*.

4.3.2 iRODS

Le protocole iRODS[12] étant très utilisé pour le stockage de grandes quantités de données, de par son aspect distribué et les fonctions de référencement et de recherche, il fait partie des premiers protocoles dont le support a été envisagé.

iRODS ne proposant pas d'interface en C (des interfaces sont proposées en Java et en PHP), il a été décidé de faire appel aux commandes shell iRODS. La lecture des sources d'iRODS aurait probablement permis de nombreuses optimisations, en évitant notamment la construction d'un fichier temporaire, mais ne s'avérait pas pertinent sur le long terme car le travail de rétro ingénierie du code source des commandes aurait probablement dû être recommencé lors des mises à jours d'iRODS. Il semble cependant intéressant de modifier ce module pour utiliser une interface plus complète quand elle sera disponible en C/C++.

Parmi les évolutions à apporter à ce module on note la gestion de protocoles d'authentifications tels que *Kerberos*.

Le protocole iRODS étant un protocole de stockage de fichiers, le module iRODS utilise la même base que le module Scp pour ce qui est de la sérialisation/désérialisation des données.

4.3.3 WebDav

Le protocole *WebDav* permet les communications avec de nombreux services d'hébergement de données en ligne tel que *OwnCloud*[14]. Reposant sur des services de stockage de fichiers, le modules *WebDav* doit faire appel à la même structure que le module Ssh pour ce qui est de la sérialisation/désérialisation.

À l'inverse des modules Ssh et iRODS, le module Webdav fait appel à une bibliothèque plutôt qu'à des commandes. La bibliothèque Néon[13] à été choisie pour ces possibilités notamment en terme de gestion des certificats Ssl ainsi que pour son développement actif.

Contrairement aux autres modules de *data management*, le module WebDav peut fait appel à un fichier de configuration pour modifier les options d'acceptation des certificats (voir table 4).

4.3.4 Ninf1

Suite à l'avancement de la bibliothèque, j'ai eu la possibilité de présenter mon travail à M. Hidemoto Nakada le lundi 5 août. Lui ayant fourni une archive contenant Diet, la bibliothèque et plusieurs exemples pré configurés, il a développé un module Ninf compatible avec la version Ninf1. Ce module a notamment permis de prouver l'interopérabilité de la

Table 4 Options de configuration du module WebDav

Option	Valeur par défaut	Description
NOTYETVALID	false	Accepter les certificats non encore valides.
EXPIRED	false	Accepter les certificats expirés.
IDMISMATCH	false	Le `hostname` ne correspond pas à celui indiqué dans le certificat.
UNTRUSTED	false	L'autorité qui a signé le certificat n'est pas fiable.

bibliothèque, le même code ayant été utilisé pour effectuer la même opération (multiplication de matrices) en utilisant les deux middleware DIET et Ninf.

Dans un second temps, un second module sera probablement développé pour gérer NINFG, une version plus récente reposant sur *Globus*.

4.4 Modules en projet

4.4.1 Bittorrent

Un module gérant le protocole bittorrent est actuellement en cours de développement. Une des difficultés posée par ce module est la mise à disposition des fichiers `.torrent` qui nécessite l'utilisation d'un protocole annexe (décrit par les arguments de l'URI) afin de transférer le fichier `.torrent` vers un espace partagé.

4.4.2 Autres modules

Afin d'obtenir un outil aussi polyvalent que possible, il nous semble important d'offrir un très large panel de compatibilité. Ainsi il est important de pouvoir communiquer avec l'ensemble des outils utilisés par les différents clients de la bibliothèque, qu'il s'agisse de *big data* ou de modélisation physique (cosmologie, propagation de la grippe aviaire, sismologie, etc.). Bon nombre de protocoles et de services doivent donc encore être ajoutés à la liste des fonctionnalités :

- http/ftp, module qui existait dans le prototype de la bibliothèque mais qui doit être réécrit ;
- GridFTP, variation du protocole ftp pour les applications distribuées ;
- dropBox, service de stockage de données accessibles partiellement via webDav et disposant d'une API dédiée ;
- dCache, système de stockage de données orienté *big data* gérant différents protocoles (DCAP, GsiDCAP, GridFTP) ;
- Amazon S3, service de stockage proposé par Amazon [4] et reposant sur une API dédié ainsi que sur les protocoles http et bittorrent.

Partie 5 — Gestionnaire de données

Afin de remplir l'objectif primaire de la bibliothèque qui est d'implémenter un *data manager* en suivant l'API GridRPC Data Management standardisé par l'OGF, ma réflexion s'est rapidement porté sur les moyens a mettre en place pour obtenir le comportement attendu. Cette réflexion s'est faite en plusieurs temps, allant d'une implémentation simple permettant de tester les outils de base vers une implémentation plus complète et modulaire.

5.1 Du transfert simple...

La première fonction de transfert a été implémentée selon un modèle centré sur le client de la bibliothèque. Il est ici important de noter que le client la de bibliothèque peut être aussi bien le logiciel client que les services middleware présents au niveau du SeD et qui utiliseraient les fonctionnalités de la bibliothèque pour étendre leurs capacités.

Cette première fonction de transfert fonctionnait en deux temps :

- D'abord on tente d'obtenir les données à partir des URIs. Si le transfert réussit on dispose de la donnée en local et on peut alors passer à l'étape suivante. Si le transfert échoue on essaye à partir de l'URI suivante. En cas d'épuisement des URIs, une erreur est retournée car la donnée est considérée comme inaccessible.
- On envoie les données séquentiellement vers les différentes URIs définies en sortie. Si un des transferts échoue on renvoie une erreur car la donnée n'a pas pu être envoyée vers l'un des emplacements spécifiés.

Cette première fonction n'utilisait alors pas de mécanismes asynchrones et présentait donc de grosses limitations dès lors qu'il s'agissait de transférer une donnée vers plusieurs emplacements.

4. Amazon Simple Storage Service : `http://aws.amazon.com/fr/s3/`

Cette première version de la fonction de transfert a néanmoins permis de mettre en place de nombreux mécanismes utiles par la suite, qu'il s'agisse de la gestion des URIs étendues ou de la sélection des URIs en entrée.

5.1.1 Le protocole memory

Le protocole memory est un pseudo protocole imaginé dans cadre du *data manager* GridRPC Data Manager. Ce n'est pas à proprement parler un protocole puisqu'il ne s'agit pas d'un format clairement défini pour le transfert de données mais plus d'un méta protocole dans le sens où il s'agit d'un mécanisme plus haut niveau qui repose sur d'autres protocoles utilisés comme intermédiaires. L'idée derrière le protocole memory et de désigner des données en espace mémoire (comme des pointeurs). Cela suppose d'imaginer la mémoire vive des différentes machines comme un espace partagé permettant de copier des données d'une machine à l'autre.

Ce méta protocole est désigné en indiquant memory en début d'URI :

```
memory://graal.ens-lyon.fr/matriceA
```

Un tel système n'existant pas, il s'agissait de mettre en place des mécanismes permettant d'obtenir ce comportement en se reposant sur différents protocoles de transferts de données. Pour l'instant le protocole utilisé est spécifié dans le fichier de configuration de la bibliothèque mais cela est amené à évoluer rapidement, une idée étant d'utiliser un protocole passé comme paramètre de l'URI selon le format suivant :

```
memory://graal.ens-lyon.fr/matriceA?protocol=ssh
```

Ce système est divisé en différents éléments :

- Un système de mapping permettant de faire l'alias entre des chaînes de caractères décrivant une donnée local, en réalité son URI, et des pointeurs mémoires locaux. Ce système est mis en place au niveau du `grpc::Core` mais devrait probablement être déplacé vers un gestionnaire de catalogue plus complet (voir section 9.5) ;
- Un système permettant la mise à disposition des données depuis l'extérieur, reposant sur un protocole existant. DAGDA semble adapté mais pose des problèmes de déploiement car il reste jusqu'à présent lié à DIET. bittorrent ou SSH sont envisageables ;
- Des méthodes de téléchargements et d'envois automatiques des données.

Le problème principale d'un tel système est la cohérence des données, d'où l'importance de limiter les alias et d'assurer leur mise à jour, chose qui est pour l'instant compliquée avec DAGDA.

En effet si un client de la bibliothèque veut écrire une donnée qui devrait techniquement être en mémoire sur un autre client, il faut mettre la donnée à disposition sur le *data manager* utilisé comme intermédiaire et prévenir le client en question que la donnée dont il dispose n'est pas à jour (afin qu'il tente de récupérer la nouvelle donnée pour écraser l'ancienne version dont il dispose). Même si le premier mécanisme est bien mis en place, le second est plus complexe et ne pourra être entièrement fonctionnel qu'avec l'existence d'un catalogue de données et méta données local sur chaque nœud. Sans un tel catalogue il est impossible de mettre à jour l'ensemble des méta données pointant vers donnée en question.

L'idée générale de ce méta protocole est reprise dans le protocole `file` qui permet de simplifier les manipulations de fichiers, notamment en se passant d'appels aux commandes `scp` et `rsync` lors des copies/déplacements de fichiers locaux.

5.1.2 Communications asynchrones

Une fois la première fonction de transfert testée, le première amélioration a été de mettre en place les mécanismes de communications asynchrones. Pour cela le cœur de la fonction de transfert simple a été placé dans une requête appelé `grpc::IORequest` et chargée de prendre en charge le téléchargement et l'envoi des données. Ces opérations de téléchargement et d'envoi des données ont à leur tour été placées dans des requêtes de type `grpc::IRequest` et `grpc::ORequest` toutes deux construites par le `grpc::IOManager`. Cela permet donc la parallélisation des transferts tout en limitant l'encombrement de la bande passante par un trop grand nombre de requêtes concurrentes.

5.2 ... au transfert point à point

Toute la force de la bibliothèque consiste dans l'utilisation des interfaces avec les différents middleware pour construire un *data manager* capable d'effectuer des opérations sur différents serveurs et ainsi de répartir la charge afin d'assurer le passage à l'échelle nécessaire à la gestion de grosses quantités de données.

Une fois les communications asynchrones gérées, il a fallu étudier les fonctionnalités nécessaire à la distribution des opérations de transferts de données.

5.2.1 Détection du middleware

Avant de pouvoir distribuer les opérations de transferts, il est nécessaire d'identifier les nœuds pouvant participer. Cela est fait en tentant d'appeler, via les différents middleware pris en change, une fonction echo mise à disposition par la bibliothèque et devant être référencé par le middleware pour rendre ce dernier identifiable. Afin de réduire les appels vers des middleware, le choix a été fait de ne tenter de contacter que les serveurs sur lesquels la ressource est disponible à travers le pseudo protocole memory.

Une fois un serveur étant identifié comme disposant d'un middleware ainsi que des services proposés par la bibliothèque, il est alors possible de l'utiliser pour distribuer les tâches de transferts.

5.2.2 Dispatcher

Une fois les middleware identifiés, les tâches de transfert sont effectivement distribuées par un objet appelé `grpc::Dispatcher`. Le dispatcher permet de décomposer le transfert en différentes tâches élémentaires à l'aide d'un algorithme cyclique (voir figure 6).

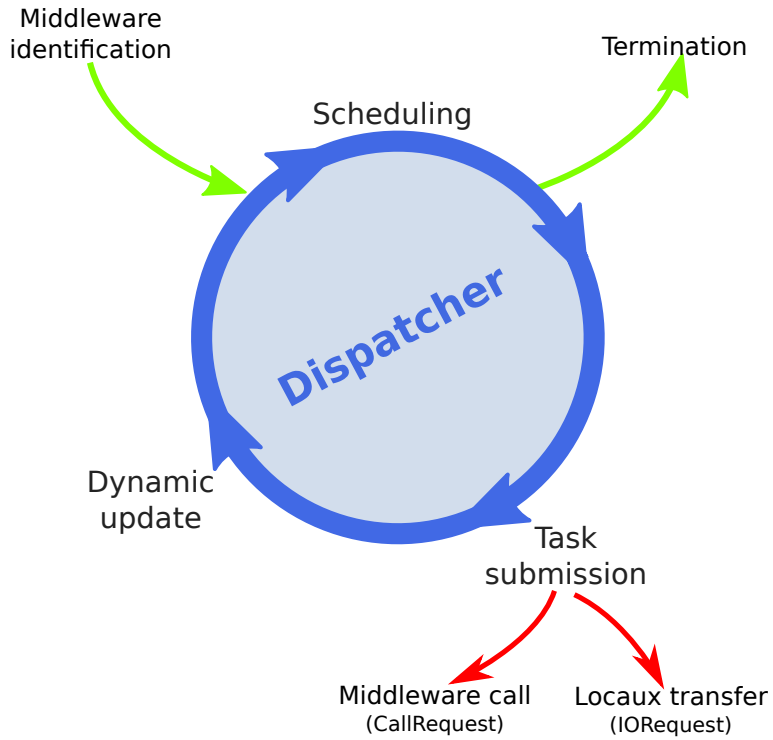

FIGURE 6 – Cycle du dispatcher

Une fois les middleware identifiés, les différentes URIs impliquées dans le transfert sont triées selon qu'elles sont locales, sur un serveur disposant d'un middleware ou sur un serveur ne disposant pas de middleware (URI distante). Une fois ce tri effectué, un algorithme d'ordonnancement se charge, selon les différents cas, de construire les tâches élémentaires adaptées et de contrôler leur exécution. Une fois les tâches élémentaires exécutées, le dispatcher met à jours les différentes listes et recommence l'opération tant que le transfert n'est pas terminé.

L'algorithme d'ordonnancement du dispatcher se charge de construire les tâches élémentaires en associant les URIs en entrée et celles en sortie suivant les différents cas décrits dans la table 5.

TABLE 5 Différents cas pouvant se présenter au niveau du dispatcher (par ordre de priorité)

		Input		
		Local	Middleware	Remote
Output	Local	3	5	2.a
	Middleware	4	6	2.b
	Remote	1.a	1.b	7

1.a Envoi d'une donnée locale sur un serveur distant.
L'opération est effectuée localement à l'aide d'une `IORequest`.

1.b Envoi d'une donnée présente sur un middleware vers un serveur distant.
L'opération est effectuée depuis le middleware en appelant le service de transfert via une `CallRequest`.

2.a Récupération d'une donnée distante.
L'opération est effectuée localement à l'aide d'une `IORequest`.

2.b Récupération sur un middleware d'une donnée distante.
L'opération est effectuée depuis le middleware en appelant le service de transfert via une `CallRequest`.

3 Transfert local (copie) d'une donnée.
L'Opération est effectuée localement à l'aide d'une `IORequest`.

4 Envoi d'une donnée locale vers un middleware.
Dans ce cas on met à disposition la donnée locale sur un *data manager* intermédiaire à l'aide d'une `IORequest`. La donnée sera ensuite récupérée par le middleware.

5 Récupération d'une donnée présente sur un autre middleware.
Dans ce cas on demande, via une `CallRequest`, au middleware de mettre la donnée à disposition puis on la récupère depuis le *data manager* à l'aide d'une `IORequest`.

6 Transfert de données d'un middleware à un autre.
On contacte, via une `CallRequest`, le middleware source pour qu'il mette la donnée à disposition. Le middleware destination y aura alors accès quand il en aura besoin.

7 Transfert de données distantes en l'absence de middleware.
On effectue toutes les opérations localement à l'aide d'une `IORequest` (transfert simple).

Dans les différents cas décrits ci-dessus, un ordonnancement est fait dans les cas où il y a plusieurs sources et/ou destinations. Cette ordonnancement, qui associe par exemple (cas 1.*b*) les middleware sources et les URIs distantes, suit un modèle de type *round robin*. Il est cependant envisager de multiplier les options d'ordonnancement à ce niveau, que ce soit à l'aide d'un ordonnanceur paramétrable ou à l'aide de modules d'ordonnancement.

Une fois les transferts lancés, on attend les résultats et on remet dynamiquement à jour les listes d'URIs. En cas de réussite, les URIs en sortie deviennent des sources valides et sont donc transférées dans la liste correspondante des URIs en entrée (on conserve alors l'information locale/middleware/distante). En cas d'échec, le middleware participant à la tâche est considéré comme non fiable et est déclassé de telle sorte qu'il ne soit plus utilisé pour le transfert de cette donnée. Le transfert sera alors confié à un autre serveur.

Si une erreur a lieu alors que l'on est dans le cas 7 (cas de base), la transfert n'a pas pu être fait et le dispatcher renvoi une erreur.

L'algorithme s'arrête quand il n'y a plus d'URIs en sortie.

5.3 Transferts implicites

En plus des transferts effectués à la demande du client par un appel à la fonction `grpc_data_transfer`, certaines circonstances peuvent amener à effectuer des opérations de transferts implicites.

5.3.1 Transferts depuis un data manager *interne du GridRPC middleware*

Nous avons vu dans la section 5.1.1 que le protocole memory pouvais être amené à mettre à disposition auprès d'un middleware des données qui auraient théoriquement du être écrites dans la mémoire d'un serveur disposant d'un middleware.

Le serveur cible n'ayant pas la donnée en mémoire, lorsqu'il tentera d'accéder au mapping mémoire relatif à cette donnée[5] ce dernier ne sera pas présent. Dans de telles circonstances on tente alors de récupérer la donnée au niveau du *data manager* dédié. Une fois ce transfert effectué, le mapping est fait pour finaliser la mise à disposition locale de la donnée.

5.3.2 Transferts automatiques lors des appels

Lors d'un appel à un service middleware, plusieurs méta données sont sérialisées pour être transmises au service distant. Ces méta données décrivant un ensemble d'URIs, certains cas peuvent amener à effectuer des transferts de données additionnels avant et après l'exécution du service (voir figure 7).

– Si l'une des URIs en entrée correspond à une donnée disponible sur le client via le protocole memory (ou file), cette donnée doit être mise à disposition avant l'appel au service de telle sorte que le service en question puisse y accéder à distance.

5. Le mapping fait le lien entre l'URI et l'adresse de debut du bloc memoire décrit

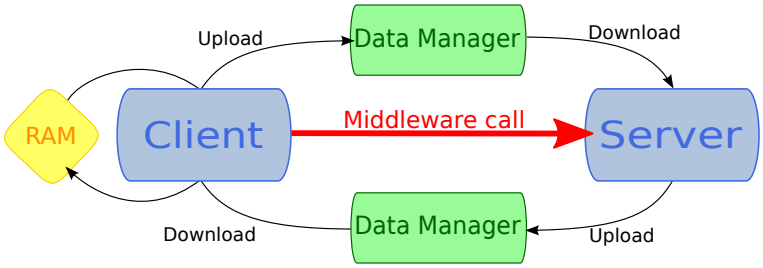

FIGURE 7 – Transferts automatiques lors des appels

– Si l'une des URIs en sortie correspond à une donnée à placer sur le client via le protocole memory (ou file), cette donnée a dû être mise à disposition par le service et doit donc être récupérée par le client. On évite ainsi d'avoir par la suite à effectuer un transfert implicite tel que décrit dans la section 5.3.1.

Ces deux cas sont intégrés aux requêtes gérant les appels middleware et sont donc effectués de manière transparente et asynchrone. Le requêtes relatives à un appel middleware ne sont pas considérées comme terminées tant que l'ensemble des transferts post-appel n'est pas fini.

Partie 6

Travaux annexes

6.1 Génération du code

La compilation de la bibliothèque est faite en utilisant l'outil `Cmake`. Plusieurs options permettent d'activer ou de désactiver la compilation des modules et des différents exemples. L'appel à `Cmake` est simplifié par un script `./generator.sh` qui permet une installation entièrement automatique. Cet outil s'est révélé particulièrement utile lors de la distribution de la bibliothèque auprès de contributeurs extérieurs ne maîtrisant pas le cœur de la bibliothèque.

Dans la dernière version, actuellement distribuée avec la bibliothèque, les outils de génération automatique configurent également différents scripts permettant par la suite de lancer automatiquement les différents exemples.

6.2 Déploiements et tests

Au cours du développement de la bibliothèque, les différentes fonctionnalités ont été régulièrement testées à l'aide d'expériences dédiées. Tous ces exemples d'utilisation, initialement séparés du reste des sources de la bibliothèque, ont pour certains été intégrés à un dossier `testing` (voir section 3.1).

Ces différents exemples, tous documentés, permettent aux futurs utilisateurs d'étudier les exemples types d'utilisation, facilement exécutables à l'aide de scripts dédiés.

Les sources des différentes expériences présentées dans la section 8 sont eux aussi disponibles

6.3 Documentation

Afin de simplifier le travail de toute personne désirant contribuer au développement de la bibliothèque, il est primordial de bien documenter tout le travail effectué.

Le travail de documentation effectué se décompose en plusieurs parties :

– Un guide décrivant les différentes étapes à suivre pour l'écriture de nouveaux modules ;
– Les commentaires présents dans le code, et notamment au niveau des composants complexes tels que le framework de requêtes asynchrones. L'un des objectifs est de prendre en charge l'outil de génération automatique de documentation *Doxygen* ;
– Des fichiers texte de documentation des différents exemples présents dans le dossier `testing`.

6.4 Transmission de connaissances

Une autre composante de mon stage a été la transmission de connaissances. Le 5 août 2013, j'ai eu la chance de rencontrer M. Hidemoto Nakada[6] pour lui présenter les différentes possibilités offertes par la bibliothèque ainsi que la structure des différents modules.

6. `https://sites.google.com/site/hidemotonakada/`

Cette rencontre a permis le développement d'un module assurant l'intégration de Ninf1, une version minimaliste du middleware Ninf, dans la bibliothèque, que M. Hidemoto Nakada nous a présenter lors d'une seconde rencontre le 15 août 2013.

À l'avenir, un second module est amené à être évoluer pour permettre de gérer NinfG, une version plus récente de Ninf reposant sur *Globus*.

Partie 7

Bilan

7.1

Contraintes et difficultés rencontrées

7.1.1 Contraintes liées à la construction modulaire

La structure modulaire présente de nombreux avantages mais a aussi certaines limites liées à la structure interne du code.

Une fois compilé, le module devient un fichier en code assembleur qui est chargé dynamiquement pour faire partie du code du programme. L'accès aux différentes fonctions se fait à l'aide de pointeurs de fonctions allouées dynamiquement et pointant vers les adresses mémoires des fonctions dans le fichier assembleur du module.

Ainsi, si l'on charge deux fois le même module, les pointeurs de fonctions obtenus sont les mêmes car pointent vers les mêmes données. Cela pose un problème vis-à-vis des données statiques. Les emplacements mémoires réservés aux données statiques étant les mêmes, il n'est par exemple pas possible de charger deux fois le même module pour l'initialiser avec des données différentes.

Un moyen de contourner cette limitation serait de construire les modules comme des *daemons* afin que chaque module soit un processus à par entière et dispose donc de ses propres données statiques.

7.1.2 Contraintes liées aux standards

La combinaison des APIs GridRPC et GridRPC Data Management implique que les arguments des fonctions `grpc_call` et `grpc_call_async` ne soient plus des `VA_ARGS` mais une liste de `grpc_data_t*`. Les méta données étant transmises sous forme sérialisées cela revient à dire que les seules communications supportées de manière standard sont l'envoi de chaîne de caractères.

Les services exécutés peuvent donc renvoyer, de manière standard, un simple entier (code d'erreur). Toute autre donnée doit être transférée via la fonction de transfert puis récupérée coté client, ce qui est non seulement consommateur de ressources mais demande de réserver des emplacements spécifiques.

La récupération de données précises, notamment pour ce qui est du statut des opérations de transfert est donc très limitée.

L'ajout d'une chaîne de caractères de retour permettrait de transmettre ce genre d'information mais complexifierait l'écriture de services middleware compatibles. Cette évolution est envisagée mais pose toujours plusieurs questions.

7.1.3 Contraintes liées aux outils utilisés

iRODS Dans le cadre de du développement du module iRODS, j'ai dû installer un serveur iRODS, nécessaire pour tester le module associé.

Malgré tous les outils d'installation automatique proposés avec iRODS, il s'avère que la connexion à *postgres* pose problème sous Ubuntu, certaines bibliothèques (libodbcpqsl) ayant changé de nom.

Grid 5000 Lors de la validation des mécanismes mis en place, l'expérience 2 à été conçue pour être déployée sur un réseau disposant d'un grand nombre de machines.

Le choix a donc été fait de la déployer sur Grid 5000. De nombreuses difficultés ce sont cependant posées lors de l'installation de la bibliothèque (voir section 8.2). LA solution aux différents problèmes rencontrés de passer par le déploiement d'images système. Le manque de temps ne nous a cependant pas encore permit de déployer cette solution.

7.2

Acquisition et approfondissement de nouvelles connaissances

Au cours de ce stage, j'ai eu l'occasion d'approfondir mes connaissances dans de nombreux domaines.

D'un point de vue purement technique, le développement de la bibliothèque m'a permis d'approfondir ma maîtrise du C++11 et de la programmation orientée objet ainsi que des mécanismes de multithreading. J'ai par ailleurs été amené à étudier la construction d'un programme modulaire que se soit au niveau de la compilation des modules comme

de leur chargement dynamique. J'ai également eu l'occasion d'utiliser l'outil `Cmake` pour la construction d'un projet présentant un nombre important de dépendances.

Ce stage a également été l'occasion de mieux appréhender la recherche dans les domaines des applications distribuées, qu'il s'agisse des impératifs de performances comme des outils employés. J'ai à ce titre été amené à utiliser des outils très variés :

- Middleware : Diet, Ninf ;
- Gestionnaire de données : iRods, OwnCloud, Dagda ;
- Services de nommage : OmniOrb ;
- LRMS : Oar (sur Grid 5000).

Enfin se stage a été pour moi une occasion unique de contribuer au développement d'un outil correspondant aux attentes de toute une communauté et répondant à des spécifications très précises car standardisées. À ce titre, j'ai été amené à partager l'avancement de mon travail avec d'autres chercheurs du même domaine, à la fois pour les tenir au courant de l'avancée du projet et pour les intégrer au développement des modules spécifiques.

Partie 8

Validation

8.1

Expérience 1 : validation des transferts multi protocolaires

L'expérience 1 à été conçue pour montrer l'impact de la localité des données sur les performances de transfert et justifier les mécanismes de préférences mis en place.

Cette expérience consiste à récupérer deux matrices disponibles via le protocole SSH, les additionner, et renvoyer le résultat sur un serveur iRODS local. Le choix de l'addition a été fait de manière à réduire le temps de calcul (par rapport à une multiplication) et ainsi se concentrer sur les temps de transferts.

Dans le cas présent on s'intéresse surtout au temps de latence, le temps de transfert étant réduit par la faible taille des matrices (512 octets). Ainsi pour de grosses matrices, l'influence du débit devrait amplifier le comportement observé dans les résultats.

Le protocole consiste à répéter l'opération en modifiant les informations décrivant les données (ajout et suppression d'URI). L'objectif est de voir l'impact du choix fait par la bibliothèque au moment de la récupération de données, les données locales (disponibles sur le serveur procédant au transfert) étant préférées aux données distantes.

TABLE 6 Différentes machines impliqués dans l'expérience 1

Machine	Services	Données
Arcterix (JFLI - Japon)	dietAgent, client	matA (disponible via SSH)
yume (JFLI - Japon)	service dgemm, iRODS	matA, matB (disponible via SSH)
graal (Ens de Lyon - France)		matB (disponible via SSH)

Dans un premier temps le client n'a pas connaissance des ressources présentes sur *yume*, et le service doit donc récupérer les données sur les machines distantes). Par la suite on répétera l'expérience en ajoutant les URIs relatives à *yume*, ce qui permettra au service de choisir automatiquement les données locales (indépendamment de l'ordre des URIs).

La figure 8 permet de visualiser les durées des différents transferts.

8.2

Expérience 2 : validation des services middleware pour la distribution des transferts de données

L'expérience 2 a pour objectif de démontrer la distribution des transferts sur les différents middlewares impliqués.

L'idée est d'envoyer une donnée, disponible sur un serveur source S vers plusieurs serveurs destination $D_1, \ldots, D_n$ disposant des services de la bibliothèques. Cette expérience, nécessitant un nombre important de nœuds disposant de middleware capables de communiquer entre eux aurait du être déployée sur la plate-forme de recherche *Grid 5000*.

Les outils disponibles sur Grid 5000 étant insuffisant, il est pour l'instant impossible d'utiliser la bibliothèque sur les nœuds réservés :

- La version de gcc disponible est la 4.4 alors que la bibliothèque nécessite la version 4.7 ou plus récente pour compiler le standard C++11 utilisé par la bibliothèque ;
- La bibliothèque Boost (nécessaire à la compilation de la bibliothèque) n'est pas disponible ;
- OmniOrbs (utilisé par DIET) n'est pas disponible.

Face aux difficultés rencontrées lors de la compilation d'une version récente de gcc, la meilleure solution semble être de déployer des images spécifiques. Le déploiement de cette expérience n'étant pas terminé, nous n'avons à l'heure actuelle pas de résultats chiffrés permettant d'évaluer l'efficacité de la distribution des transferts.

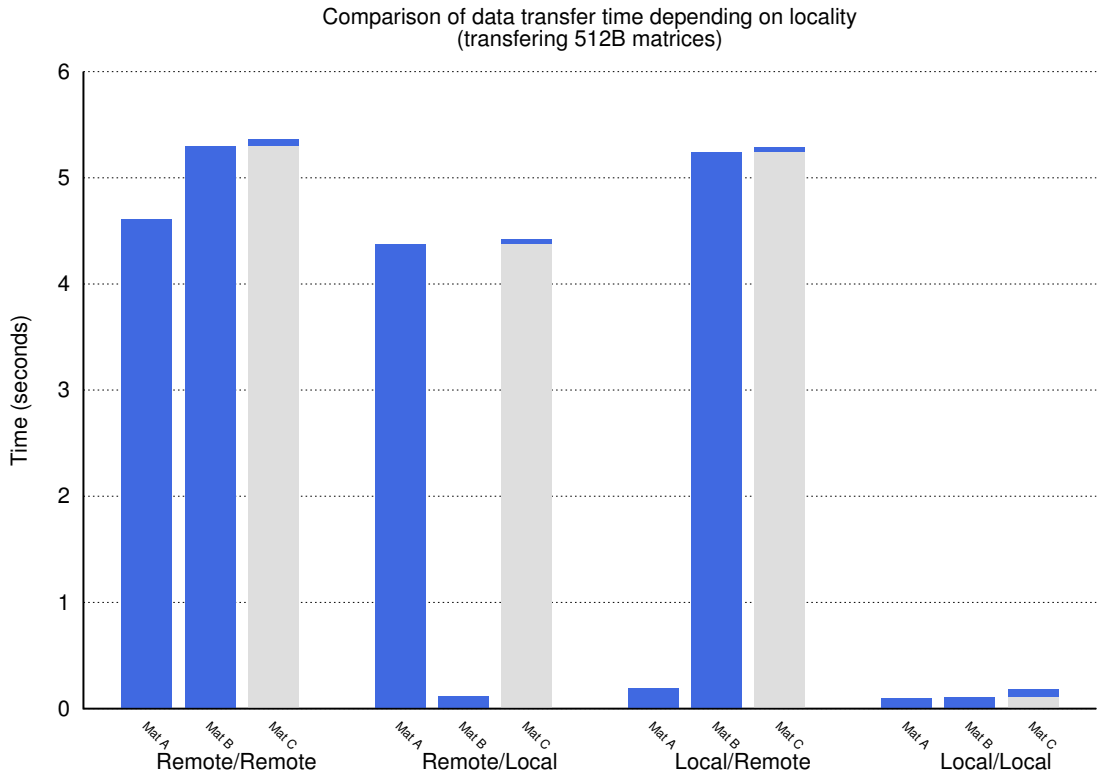

FIGURE 8 – Expérience 1 : Résultats

8.3

Expérience 3 : impact de la limitation des transferts parallèles

L'expérience 3 a pour objectif d'étudier l'impact de la limitation des transferts parallèles.

Cette expérience consiste à lancer le téléchargement de 15 matrices de 32Mb chacune et à effectuer une opération sur les différentes matrices téléchargées. Dans cette expérience on suppose que l'opération est commutative et que l'on n'est pas dans l'obligation d'attendre les matrices dans l'ordre. Il est cependant important de noter que l'opération étant supposé itérative, il est nécessaire d'attendre la fin du traitement d'une matrice avant de pouvoir traiter la suivante.

On utilise ici le mécanisme *wait any* qui permet de faire une attente sur un ensemble de transferts et de récupérer l'identifiant de la première donnée transférée.

Durant cette expérience, les données (des matrices de 2000×2000 entiers codés sur 64bits) sont accessibles via le protocole SSH. L'objectif est d'observer l'influence de la limitation du nombre de téléchargements parallèles, qui variera de 1 à 8. Les résultats sont présentés dans la figure 9.

Partie 9

Perspectives d'évolution

9.1

API

`grpc_get_info` Lors de la standardisation de l'API gridRPC Data Management, la fonction `grpc_get_info` a été prévue afin de permettre aux clients de la bibliothèque d'obtenir des informations sur l'état des transferts en cours.

Le comportement de cette fonction n'a cependant pas été précisé à aucun moment.

L'implémentation de cette fonction est donc suspendue à la spécification des différentes informations que cette fonction doit pouvoir fournir ainsi qu'au format de retour attendu.

Tant que ces choix n'auront pas été fais, l'utilisation de la bibliothèque restera limitée à des développeurs avertis et risque d'avoir du mal à s'imposer auprès des utilisateurs potentiels.

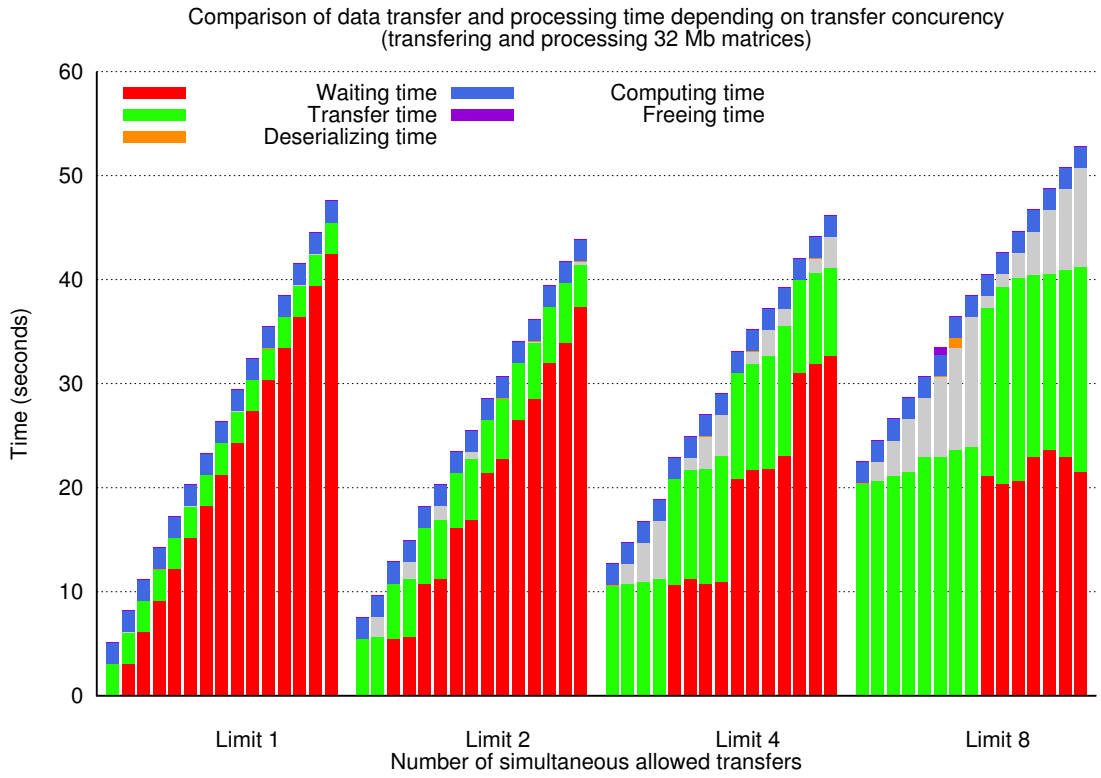

FIGURE 9 – Expérience 3 : Résultats

9.2 Protocoles

9.2.1 Protocole `memory` multiples et adaptatif

Le protocole memory permet déjà de mettre en place des mécanismes complexes de data management reposant sur DAGDA. Un des objectifs d'évolution est de rentrer ces mécanismes compatibles avec d'autres protocoles de transferts de données afin de s'adapter aux capacités des différents nœuds.

Le passage de certaines options à travers les arguments de l'URI devrait ainsi permettre d'augmenter le caractère polyvalent de ce méta protocole.

9.2.2 Protocole `file`

Le protocole file est un méta protocole proche du protocole memory en raison de sa conception. Son développement permettra d'améliorer les capacités de gestions des fichiers de données, ce qui étoffera encore les possibilités offertes par le data manager de la bibliothèque.

L'une des amélioration les plus visibles sera de gérer de manière plus efficace les données et fichiers de données afin d'éviter certaines recopies lors de transferts impliquant des services de stockage de fichiers.

9.2.3 Transfert SSH point à point

Lors du transfert de fichiers entre deux services de stockage de données, la méthode actuelle consiste à télécharger les données depuis la source puis à les envoyer sur le serveur cible. Certains services, SSH notamment, permettent cependant d'effectuer un transfert point à point à l'aide d'une connexion distante ce qui présente de nombreux avantages. Ces modes de transferts, non supportés à l'heure actuelle, devront à l'avenir être intégrés aux modules et gérés par les dispatcher. L'une des solutions envisagées serait de construire un système middleware basé les des commandes shell exécutées via SSH.

9.3 Ordonnancement

9.3.1 Algorithmes pour le dispatcher

Pour coupler les URIs et ainsi former des tâches de transferts exécutables de manière distribuées, le dispatcher utilise actuellement un ordonnanceur simple. Afin d'améliorer la modularité de la bibliothèque, cette partie devra être modifiée de manière à être modulaire. Un ordonnanceur standard et configurable sera alors fourni de base, tout en laissant la possibilité à chacun de développer son propre module d'ordonnancement afin de répondre à ses besoins spécifiques.

9.3.2 Ordonnancement tâches/transfert

En l'état actuel, les soumissions des tâches de calculs et les transferts sont découplés. Cela limite grandement les possibilités d'ordonnancement des transferts, le data manager n'ayant pas connaissance des calculs dans lesquels interviennent les données.

Il sera donc, à terme, nécessaire d'offrir une interface plus large permettant de fournir des DAGs complets à partir desquels on pourra faire le co-ordonnancement des tâches de calculs et des transferts de données. Cela permettra notamment le prefetch de données.

9.4 Évaluation des performances

Afin d'améliorer les performance des mécanismes d'ordonnancement, il semble intéressant de conserver des informations des précédents transferts. Pour cela il faudrait ajouter un gestionnaire de performance chargé de maintenir des statistiques sur les différentes communications réalisées. Ce gestionnaire devra de plus être capable d'évaluer les nouvelles connexions entre deux serveurs distants afin d'aider à faire le choix entre une connexion déjà utilisée et connue et une connexion jusqu'alors inutilisée.

La mise en place d'un gestionnaire de performances réseau devrait aussi permettre de réévaluer en temps réel le nombre maximum de connexions simultanées.

9.5 Catalogue de données

La cohérence des données manipulées par le data manager est généralement assurée par le gestionnaire de donnée sous-jacent. Cependant, dans le cas des protocoles memory et file, la bibliothèque doit se charger des communications entre différentes machines et différents middleware. À ce titre la bibliothèque se doit de garder des informations sur les différentes données gérées afin de gérer la consistance des différentes copies confiées à différents gestionnaires de données.

Un tel catalogue est aussi nécessaire pour obtenir des informations sur les transferts en cours sur différents nœuds et concernant une même donnée. Les catalogues présents sur tous les nœuds devront ainsi être capable de communiquer à l'aide d'appels à des services dédiés mis en place par bibliothèque et référencés par les middlewares.

Partie A

Annexes

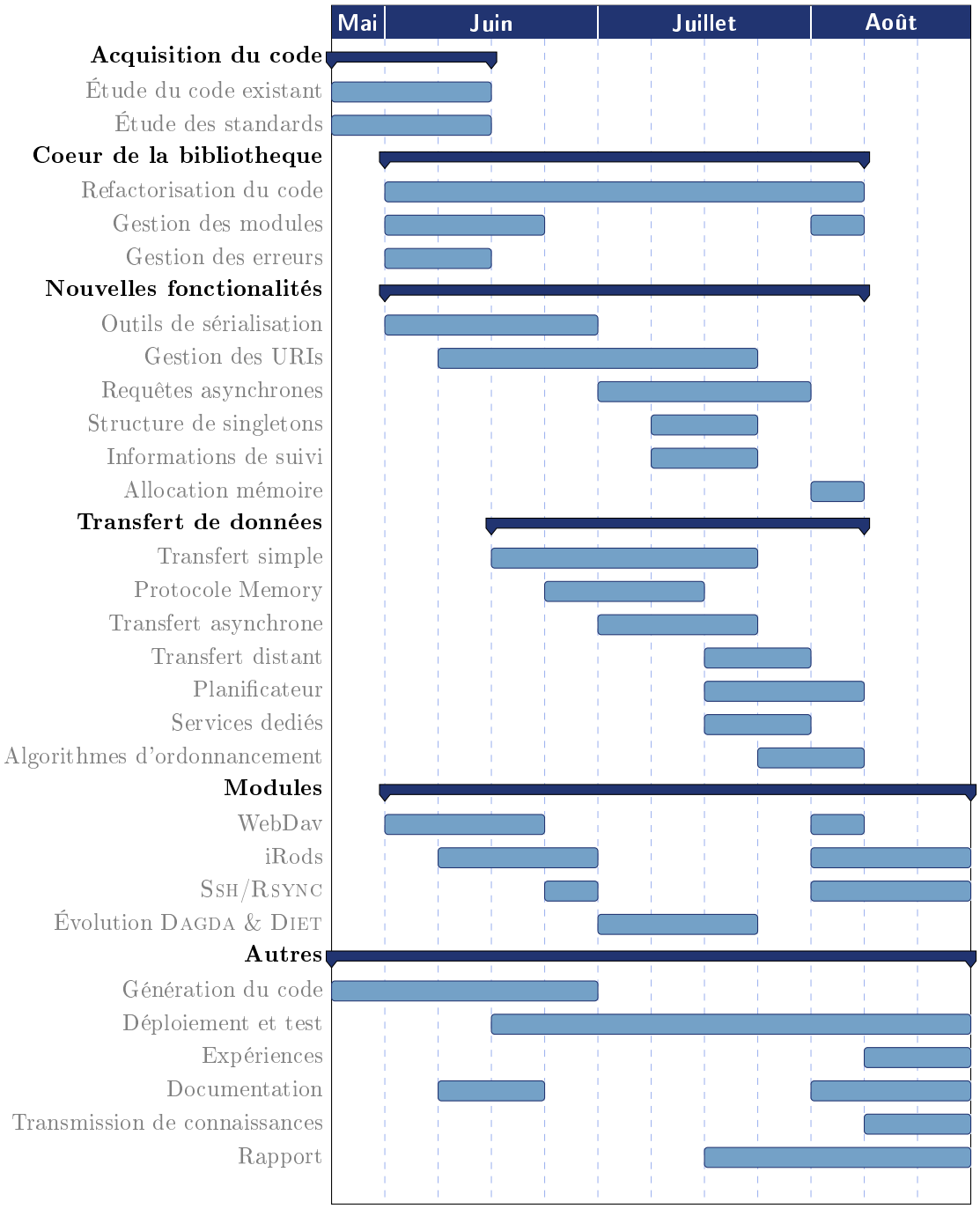

Figure 10 – Planning du stage

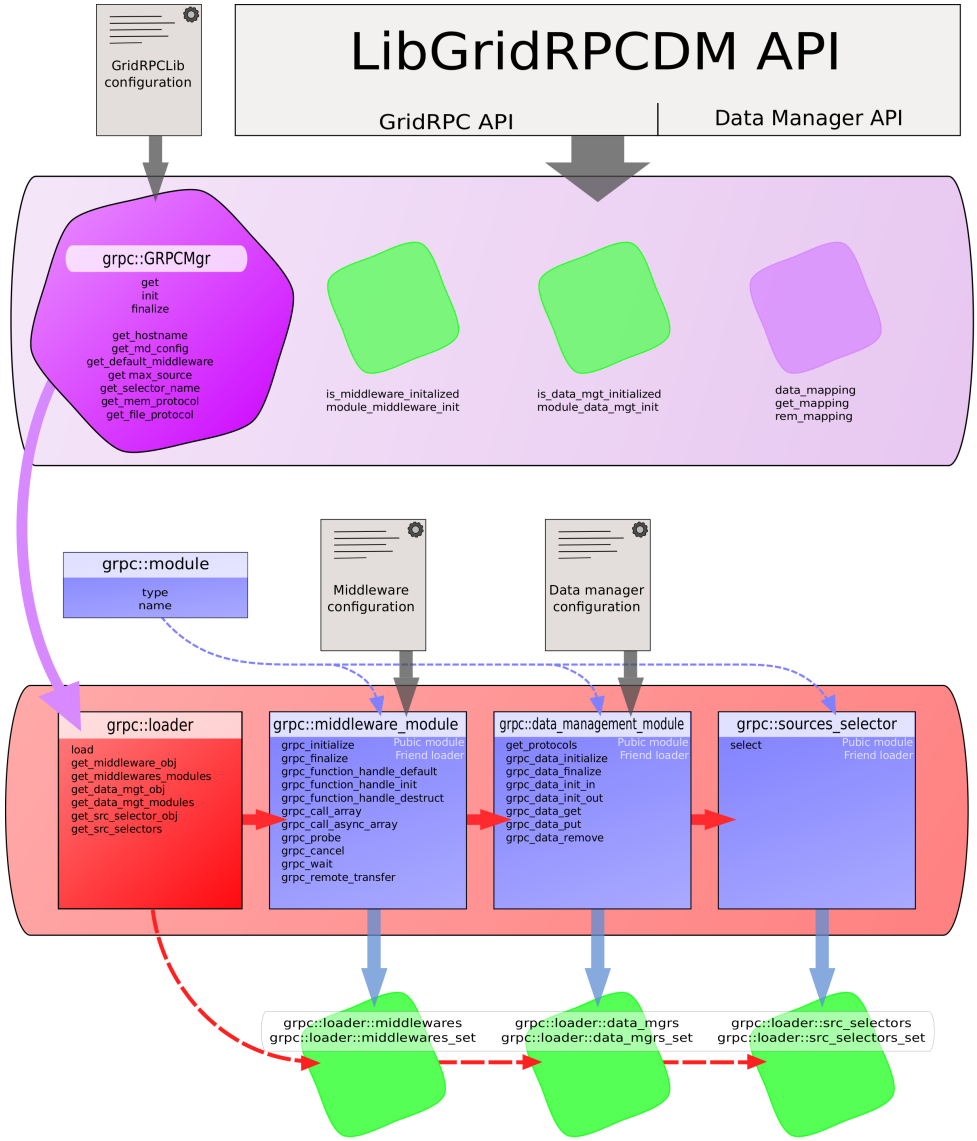

FIGURE 11 – Diagramme de la procédure d'initialion des modules

Références

[1] Asim Yarkhan, Jack Dongarra, and Keith Seymour. GridSolve : The Evolution of A Network Enabled Solver. 2002.

[2] Y Tanaka, H Nakada, S Sekiguchi, T Suzumura, and S Matsuoka. Ninf-G : A Reference Implementation of RPC-based Programming Middleware for Grid Computing. pages 41–51, 2003.

[3] T Berners-Lee, R Fielding, and L Masinter. Uniform Resource Identifier (URI) : Generic Syntax. *rfc3986*, pages 1–62, 2005.

[4] Eddy Caron. DIET : A Scalable Toolbox to Build Network Enabled Servers on the Grid. 33(June), 2005.

[5] T. Kosar. A New Paradigm in Data Intensive Computing : Stork and the Data-Aware Schedulers. *IEEE Challenges of Large Applications in Distributed Environments*, pages 5–12, 2006.

[6] H Nakada, S Matsuoka, K Seymour, J Dongarra, C Lee, and H Casanova. A GridRPC Model and API for End-User Applications. *GFD-R.052, GridRPC Working Group*, 2007.

[7] Y Tanimura, K Seymour, E Caron, A Amar, H Nakada, Y Tanaka, and F Desprez. Interoperability Testing for The GridRPC API Specification. *GFD-E.102, GridRPC Working Group*, 2007.

[8] Y Caniou, E Caron, G Le Mahec, and H Nakada. Data Management API within the GridRPC. *GFD-R-P.186, GridRPC Working Group*, 2011.

[9] Yves Caniou, Eddy Caron, and Hidemoto Nakada. Standardized Data Management in GridRPC Environments. *ICCIT*, 2011.

[10] Yves Caniou, Eddy Caron, Le Mahec, and Hidemoto Nakada. Transparent Collaboration of GridRPC Middleware using the OGF Standardized GridRPC Data Management API. *PoS ISGC*, 2012.

[11] Inpt Enseeiht. Evaluation of the OGF GridRPC Data Management library, and study of its integration into an International Sparse Linear Algebra Expert System. *PoS ISGC*, pages 1–12, 2013.

[12] IRODS. IRODS : Data Grids, Digital Libraries, Persistent Archives, and Real-time Data Systems.

[13] Neon. neon HTTP and WebDAV client library.

[14] OwnCloud. ownCloud.org | Your Cloud, Your Data, Your Way !

Liste des tableaux

Liste des exemples de code

Liste des figures