Grid RPC システムの API の提案

中田秀基[†,††] 田中良夫[†]
松岡聡[††,†††] 関口智嗣[†]

RPC(Remote Procedure Call) に基づく計算システムは、広域分散計算環境である Grid 上のミドルウェアの一形態として有望である。Grid 上の RPC システムは、Ninf、Netsolve などいくつか提案されている。しかし API に標準規格が存在しないため、RPC システムを用いて記述したプログラムに互換性がなく、このことが Grid 上の RPC の普及を妨げている。本稿では、Grid 上の RPC システムの標準 API の候補として、一つの API を提案する。この API は、数年にわたる Ninf システムにおける経験に基づき、必要十分の機能を提供しながら、最小限となるように設定されている。我々は Global Grid Forum などの場で、この規格案の標準化を促進していく予定である。

A Proposal for API of GridRPC

Hidemoto Nakada,[†,††] Yoshio Tanaka,[†] Satoshi Matsuoka[††,†††]
and Satoshi Sekiguchi [†]

Computation system based on RPC(Remote Procedure Call) is a promising candidate as a middleware of the Grid. Several systems, including Ninf and NetSolve, are already proposed and used in various area. However, Grid RPC API is not standardised yet, and the fact is precluding further spread of Grid RPC systems. In this paper, we examine two existing Grid RPC API and propose a Grid RPC API intended to be a standard, based on them. The API is designed to be minimal but sufficient for applications. We are planning to promote this API as a standard for Grid RPC in Global Grid Forum.

1. はじめに

RPC(Remote Procedure Call) に基づく計算システムは、広域分散計算環境である Grid 上のミドルウェアの一形態として有望である。Grid 上の RPC システムは、Ninf[1)]、Netsolve[2)] などいくつか提案されており、それぞれさまざまな分野で実用的に使用されつつある。これらの RPC システムは本質的に等価であるにもかかわらず、個々の API が異なるため、これらの RPC システムを用いて記述したプログラムには互換性がなく、このことが、この種のシステムのさらなる普及を妨げている。

RPC システムを利用したアプリケーションプログラムがシステム間で互換性を保つための方法としては、1) システムの相互運用を可能にするブリッジを作成する、2)API を共有化する、ことが考えられる。

われわれはかつて、前者のアプローチに従って、NetSolve と Ninf の相互運用を可能にするブリッジを提案・開発した[3)]。このブリッジを用いると Ninf システムのサーバ群を NetSolve のサーバとして、NetSolve のサーバ群を Ninf のサーバとして使用することができる。このため、同じプログラムで双方のサーバにアクセスできるだけでなく、同時に双方のサーバを使用することができる。しかし、ホップ数が増えることによって性能が低下すること、システムプロトコルの更新に追従してブリッジモジュールを常に更新しつづけなければならないこと、それぞれの資源管理データのコンシステンシを保つのが難しいこと、N 個のシステムに対して $\frac{N^2}{2}$ 個のブリッジが必要になることなどから、実用的ではないと判断した。

本稿では、後者のアプローチに従って API の共有化を試みるべく、Grid 上の RPC システムの標準候補となる API を提案する。この API は、1) 十分な記述力と一般性、2) 既存の RPC システムとの親和性、3) 実装の容易さ、を念頭において定義されている。

以降、2では Grid 上の RPC システムに対する要請を整理する。3では既存の RPC システムの API について述べる。4で、GridRPC の API の方針を示し、5で、本稿で提案する GridRPC の API を示す。さらに、この API を用いた典型的なプログラムを示す。

† 産業技術総合研究所 National Institute of Advanced Industrial Science and Technology (AIST)
†† 東京工業大学 Tokyo Institute of Technology
††† 科学技術振興事業団
Japan Science and Technology Corporation

2. Grid 上の RPC システム

Gird 上の RPC システム (以降 GridRPC システム) は、クライアント・サーバ型の計算システムであり、サーバに存在する計算ルーチンをクライアント側のプログラムから、容易に実行できるようにすることを目的としている。

クライアント側には、プログラマに対して API を提供し、RPC 呼び出しを行うクライアントライブラリが必要になる。

サーバ側には、提供するルーチンのインターフェイス情報記述から、スタブルーチンを生成するインターフェイス情報コンパイラ、RPC 呼び出しを受けて、スタブルーチンを起動するなんらかのサーバが必要である。

3. 既存の GridRPC システム

ここでは既存の GridRPC システムである Ninf と NetSolve について概説する。

3.1 Ninf

Ninf システムは旧電総研が中心となって開発した Grid 向け RPC システムである。特徴としては、以下が挙げられる。

- MetaServer と呼ぶプロキシ機構による動的負荷分散
- IDL 記述が容易
- ファイル転送に対応
- サーバ側からクライアントの関数を呼び出すコールバック機能
- サーバグループに対してスロットリングを行いながら RPC を行うグループ API 機能

3.1.1 クライアント API

表 1 に Ninf のクライアント API を示す。Ninf のクライアントは基本的に単一のサーバに対する呼び出しを指向しており、サーバ名はオプション引数や環境変数で指定する。複数サーバに対する実行を行う際にはプロキシとして機能する MetaServer をサーバとして指定することで実現する。

これでは不便なこともあるため、以下のように RPC ライブラリ名にホストとポート番号をエンコードすることで、プログラム中から直接サーバを指定することもできる。

```
ninf://SERVER_NAME:PORT/module/function
```

3.1.2 サーバ側記述

サーバ側での IDL 記述が容易なことは Ninf の特徴の一つである。基本的に C における関数のインターフェイス定義に入出力のモードを追加したものである。**図 3.1.2** に行列の乗算ルーチンを表現したものを示す。

この記述を IDL コンパイラで処理することで、make ファイルとスタブメインが得られ、make ファイルを実行することで、実行バイナリが得られる。

```
Define dmmul(mode_in  int n,
             mode_in  double A[n][n],
             mode_in  double B[n][n],
             mode_out double C[n][n])
Required "mmul.o"
Calls "C" mmul(n,A,B,C);
```

図 1 Ninf のサーバ側記述

Ninf システムではこのバイナリをサーバに登録することで外部からの RPC 呼び出しが可能になる。

3.2 NetSolve

NetSolve はテネシー大学の Jack Dongarra 教授らが開発した RPC システムである。NetSolve には以下の特徴がある。

- エージェントと呼ばれるスケジューリングモジュール
- Matlab、Mathematica などの数式処理システムから呼び出すためのインターフェイス
- 大規模並列計算のための Farming API
- 関数を動的にサーバにシップして実行する User Supplied Function 機能

3.2.1 クライアント API

表 2に NetSolve のクライアント API の一部を示す。基本的には Ninf のそれと非常によく似ていることがわかる。相違点としては、Ninf では複数の RPC の終了を待つ API が豊富に用意されているが、NetSolve では単一の RPC を待つ API しか用意されていないことがわかる。これは Farming 用の専用 API を持つため、不要と考えているためであろう。

3.2.2 サーバ側記述

サーバ側の記述は非常に複雑である。**図 3.2.2** に Linear Solver のインターフェイスを記述したものを示す。

4. GridRPC の API 策定の方針

前節の 2 システムの API を踏まえて、以下の方針で API を策定した。

4.1 API の範囲

Grid 上 RPC の真の標準化のためには、クライアント API だけでなく、サーバ側のインターフェイス記述、サーバ側、クライアント側のメンテナンス用コマンドにいたるまで標準化しなければならない。

しかし、Grid 上 RPC の機構はまだ研究段階であり、クライアント API 以外の部分は今後も変更が予想されるため、今回はクライアント API のみを対象とする。

4.2 機能の選択

Ninf のコールバック機能、NetSolve の User Sup-

表 1　Ninf のクライアント API

API 関数名	機能
int Ninf_call(char *, ...)	ブロッキング RPC 第 1 引数で RPC ライブラリ名を指定する 返り値は 成功時には NINF_OK、失敗時には NINF_ERROR
int Ninf_call_async(char *, ...)	ノンブロッキング RPC 第 1 引数で RPC ライブラリ名を指定する 返り値は 成功時には正の整数値であるセッション ID 失敗時には NINF_ERROR
int Ninf_session_probe(int)	特定のノンブロッキング RPC が実行中かどうかの検出 引数はセッション ID 返り値は真偽値を示す int
int Ninf_session_cancel(int)	特定のノンブロッキング RPC の実行中止 引数はセッション ID 返り値は成功時には TRUE、失敗時には FALSE
int Ninf_wait(int)	特定のノンブロッキング RPC の終了を待つ 引数はセッション ID 返り値は 成功時には NINF_OK、失敗時には NINF_ERROR
int Ninf_wait_all()	実行中のノンブロッキング RPC のすべての終了を待つ 返り値は 成功時には NINF_OK、失敗時には NINF_ERROR
int Ninf_wait_any(int *)	実行中のノンブロッキング RPC のいずれかの終了を待つ 第 1 引数のポインタに終了したセッションの ID を返す 返り値は 成功時には NINF_OK、失敗時には NINF_ERROR

表 2　NetSolve のクライアント API

API 関数名	機能
int netsl(char *, ...)	ブロッキング RPC 第 1 引数で RPC ライブラリ名を指定する 返り値は ステイタスコード
int netslnb(char *, ...)	ノンブロッキング RPC 第 1 引数で RPC ライブラリ名を指定する 返り値は 成功時には 正数であるのリクエストハンドル、 失敗時にはエラーコードを返す
int netslpr(int)	特定のノンブロッキング RPC が実行中かどうかの検出 返り値は NetSolveNotReady または NetSolveOK
int netslwt(int)	特定のノンブロッキング RPC の終了を待つ

plied Function 機能に関しては、システムの構造に強く依存し、用途も限られるため今回の API には含めない。

Ninf のグループ API と NetSolve の Farming API は、双方ともアドホックに実装されている感があり、未成熟であることから、今回の API には含めない。ただし、今回の API を用いれば容易に実装できるものと思われる。

4.3　サーバ選択機能

Ninf には MetaServer、NetSolve にはエージェントがあり、それぞれ自動的なサーバ選択と負荷分散を行う機能がある。

しかし、GridRPC をより高位の Grid アプリケーションのビルディングブロックとしてとらえるならば、スケジューリング機能は GridRPC に含めず、任意の機構が使用できるようにしたほうが、一般性が向上すると考えられる。このため、本 API ではサーバを明示的に指定する関数のみを用意する。

4.4　リモート関数ハンドル

RPC 呼び出しは、リモート関数のハンドルを取得し、それを用いて行う。このようにリモート関数を抽象化しておくことで将来の負荷分散機構の導入が容易になるし、システム固有の問題を吸収することが容易になる

4.5　エラーコード、ステイタスコード

エラーコード、ステイタスコードは直接プログラム中に現れるのでこれに関しても定義を行う必要がある。実際には、エラーやステイタスのバリエーションはシステムの構造に依存するため、完全な標準化は難しい。そこで、エラー、ステイタスのコードは、システムに依存しない一般的な用語を用いて定義し、実際のシステムとのマップはエラー文字列で行うことで対処する。

エラーコードは C のプリプロセッサで使用する文字列で定め、具体的な整数値は定めない。ステイタスコードは値も定める。

```
@PROBLEM linsol
@INCLUDE ¡math.h¿
@LIB -L/home/lib/
@FUNCTION linsol
@LANGUAGE FORTRAN
@MAJOR COL
@PATH LinearAlgebra/LinearSystems/
@DESCRIPTION
Solves the square linear system A*X = B. Where:
A is a double-precision matrix of dimension NxN
B is a double-precision matrix of dimension NxNRHS
X is the solution
@INPUT 2
@OBJECT MATRIX D A
Matrix A (NxN)
@OBJECT MATRIX D B
Matrix B (NxNRHS)
@OUTPUT 1
@OBJECT MATRIX D X
Solution X (NxNRHS)
@COMPLEXITY 3,3
@CALLINGSEQUENCE
@ARG I0
@ARG I1,O0
@ARG nI0,mI0,mI1
@ARG nI1
@ARG lI0
@ARG lI1,lO0
@CODE
```

図 2 NetSolve のサーバ側記述

5. GridRPC の API

5.1 インクルードファイル

インクルードファイル名は grpc.h とする。

5.2 ステイタスコードとエラーコード

ステイタスコードを表 3に、エラーコードを表 4に示す。

表 3 GridRPC ステイタスコード

コード	値	意味
GRPC_OK	0	成功と表す
GRPC_ERROR	-1	失敗を表す

5.3 イニシャライザとファイナライザ

初期化と後処理を行う関数である。

```
int
  grpc_initialize(
    char * config_file_name);
```

コンフィギュレーションファイルを読み出して、RPC システムに必要な初期化を行う。この関数が呼ばれるよりも前に、RPC 用の関数を呼び出した場合には、正常な動作は保証されない。

引数としてコンフィグレーションファイルの名前を与える。コンフィグレーションファイルの機能は、各システムの構成に依存するので、ここでは規定しない。初期化が成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す

```
int grpc_finalize();
```

RPC に使用した資源を解放する。この関数が呼ばれた以降に RPC 用の関数を呼び出した場合には、正常な動作は保証されない。引数はなし。初期化が成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す

5.4 リモート関数ハンドル

リモート関数のハンドルにはホスト情報と関数名の情報が収められる。リモート関数のハンドルを取得する関数は 2 つ用意した。

```
int grpc_function_handle_default(
    grpc_function_handle_t * handle,
    char * func_name);
```

この関数はデフォルトのホスト、ポートを使用して、第 1 引数で与えられる構造体領域に書き込む。初期化が成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す。

func_name は構造体に埋め込まれて使用されるので、呼び出しが終わるまで、上書きや解放をしてはならない。

```
int grpc_function_handle_init(
    grpc_function_handle_t * handle,
    char * host_name,
    int    port,
    char * func_name);
```

この関数は、サーバのホストとポートを明示的に指定してリモート関数ハンドルを取得する。初期化が成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す。

host_name、func_name は構造体に埋め込まれて使用されるので、呼び出しが終わるまでオーバライト、解放をしてはならない。

5.5 RPC 呼び出し

RPC 呼び出しを行う関数にはブロッキングとノンブロッキングの 2 つがある。

```
int grpc_call(
    grpc_function_handle_t *,
    ...);
```

第 1 引数で指定したハンドラを使用してブロッキングで RPC 呼び出しを行う。成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す。

```
int grpc_call_async(
    grpc_function_handle_t *,
    ...);
```

第 1 引数で指定したハンドラを使用してノンブロッキングで RPC 呼び出しを行う。成功すればその呼び出しのハンドラとなる正数、セッション ID を返す。失敗すれば GRPC_ERROR を返す。

5.6 ノンブロッキング呼び出しの操作

ノンブロッキングで呼び出した関数に対してはいくつかの操作が可能である。操作はセッション ID を用

表 4　GridRPC エラーコード

コード	意味
GRPCERR_NOERROR	エラーではない
GRPCERR_NOT_INITIALIZED	初期化がされていない
GRPCERR_SERVER_NOT_FOUND	サーバが見つからない
GRPCERR_CONNECTION_REFUSED	サーバへの接続に失敗した
GRPCERR_NO_SUCH_FUNCTION	指定した関数が存在しない
GRPCERR_AUTHENTICATION_FAILED	サーバでの認証に失敗した
GRPCERR_RPC_REFUSED	サーバで実行する権限がない
GRPCERR_COMMUNICATION_FAILED	通信に障害が起きた
GRPCERR_PROTOCOL_ERROR	プロトコルがおかしい
GRPCERR_CLIENT_INTERNAL_ERROR	クライアントの内部エラー
GRPCERR_SERVER_INTERNAL_ERROR	サーバの内部エラー
GRPCERR_EXECUTABLE_DIED	実行ファイルがなんらかの理由で落ちた
GRPCERR_SIGNAL_CAUGHT	計算中になんらかのシグナルが発生した
GRPCERR_UNKNOWN_ERROR	不明なエラーが発生した

いて行う。

```
int grpc_probe(int sessionID);
```

第 1 引数で指定する呼び出しが終了したかどうかを調べる。終了していれば 1 を、していなければ 0 を返す。調査自体に失敗すれば GRPC_ERROR を返す。

```
int grpc_cancel(int sessionID);
```

実行中の関数をキャンセルする。成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す。関数がすでに終了していた場合にも GRPC_OK が返される。

```
int grpc_wait(int sessionID);
```

第 1 引数で指定する呼び出しの終了を待つ。成功すれば GRPC_OK、失敗すれば GRPC_ERROR を返す。

```
int grpc_wait_and(
    int * idArray,
    int length);
```

複数の呼び出しがすべて終了するのを待つ。第 1、第 2 引数でセッション ID の配列とその長さを指定する。すべての呼び出しが成功すれば GRPC_OK、いずれかが失敗すれば GRPC_ERROR を返す。

```
int grpc_wait_or(
    int * idArray,
    int   length,
    int * idPtr);
```

複数の呼び出しのいずれかが終了するのを待つ。第 1、第 2 引数でセッション ID の配列とその長さを指定する。終了した呼び出しが成功していれば GRPC_OK、失敗していれば GRPC_ERROR を返す。第 3 引数に終了したセッションの ID をセットする。

```
int grpc_wait_all();
```

それまでに行ったノンブロッキング呼び出しがすべて終了するのを待つ。すべての呼び出しが成功すれば GRPC_OK、いずれかが失敗すれば GRPC_ERROR を返す。

```
int grpc_wait_any(
    int * idPtr);
```

複数の呼び出しのいずれかが終了するのを待つ。第 1、第 2 引数でセッション ID の配列とその長さを指定する。終了した呼び出しが成功していれば GRPC_OK、失敗していれば GRPC_ERROR を返す。第 1 引数に終了したセッションの ID をセットする。

```
grpc_function_handle_t *
  grpc_get_handle(
    int sessionId);
```

引数で指定したセッション ID が使用したハンドルのポインタを取得する。

5.7 エラー処理 API

```
int grpc_get_last_error();
```

最後に行った RPC 処理のエラーコードを取得する

```
int grpc_get\_error(int sessionID);
```

第 1 引数で指定されるセッション ID を持つセッションのエラーコードを取得する。セッション ID に対応するセッションが存在しない場合は -1 を返す

```
void grpc_perror(char * str);
```

第 1 引数で与えた文字列の後に、最後に行った RPC 処理のエラー情報の文字列を付加して標準エラーに出力する。

```
char * grpc_error_string(int error_code);
```

第 1 引数で指定されるエラーコードに対応する文字列を返す。

5.8 サンプルコード

この API を使用して記述した典型的なクライアントコード例を示す。モンテカルロ法を用いて PI を計算するルーチンを複数のサーバ上で実行するプログラムである。

6. おわりに

本稿では、Grid 上の RPC システムの標準の候補として API を提案した。現在、この API を実装するものとして、従来の Ninf システムの API を改変した

```
#include "grpc.h"
#define NUM_HOSTS 8
char * hosts[] = {"host00", "host01", "host02", "host03",
                  "host04", "host05", "host06", "host07"};
grpc_function_handle_t handles[NUM_HOSTS];
int port = 4000;

main(int argc, char ** argv){
  double pi;
  long times, count[NUM_HOSTS], sum;
  char * config_file;
  int i;
  if (argc < 3){
    fprintf(stderr, "USAGE: %s CONFIG_FILE TIMES \n", argv[0]);
    exit(2);
  }
  config_file = argv[1];
  times = atol(argv[2]) / NUM_HOSTS;

  /* GRPC の初期化 */
  if (grpc_initialize(config_file) != GRPC_OK){
    grpc_perror("grpc_initialize");
    exit(2);
  }
  /* ハンドルの初期化 */
  for (i = 0; i < NUM_HOSTS; i++)
    grpc_function_handle_init(&handles[i], hosts[i], port, "pi/pi_trial");

  for (i = 0; i < NUM_HOSTS; i++)
    /* ノンブロッキング呼び出しによる並列呼び出し*/
    if (gprc_call_async(&handles[i], i, times, &count[i]) == GRPC_ERROR){
      grpc_perror("pi_trial");
      exit(2);
    }
  /* すべての呼び出しの終了を待つ */
  if (grpc_wait_all() == GRPC_ERROR){
    grpc_perror("wait_all");
    exit(2);
  }
  /* PI の計算と表示 */
  for (i = 0, sum = 0; i < NUM_HOSTS; i++)
    sum += count[i];
  pi = 4.0 * ( sum / ((double) times * NUM_HOSTS));
  printf("PI = %f\n", pi);
  /* GRPC の後処理 */
  grpc_finalize();
}
```

図 3 GridRPC API を用いたサンプルコード

ものと、グローバルコンピューティングツールキット Globus[4] を用いて新たに実装した、Ninf-G と呼ぶシステムを実装している。

今後、実装したシステムを実プログラムに使用して、API が十分であるかを検証するとともに、Global Grid Forum[5] などの場で、標準化を呼びかけていく予定である。

本 API はドラフトの段階であるため、今後多くの変更が予想される。変更したものは、随時 Web ページ `http://ninf.apgrid.org/gridrpc_api` で公開していく予定である。

参 考 文 献

1) Sato, M., Nakada, H., Sekiguchi, S., Matsuoka, S., Nagashima, U. and Takagi, H.: Ninf: A Network based Information Library for a Global World-Wide Computing Infrastracture, *Proc. of HPCN'97 (LNCS-1225)*, pp. 491–502 (1997).

2) Casanova, H. and Dongarra, J.: NetSolve: A Network Server for Solving Computational Science Problems, *Proceedings of Super Computing '96* (1996).

3) Nakada, H., Sato, M. and Sekiguchi, S.: Design and Implementations of Ninf: towards a Global Computing Infrastructure, *Future Generation Computing Systems, Metacomputing Issue*, Vol. 15, No. 5-6, pp. 649–658 (1999).

4) Foster, I. and Kesselman, C.: Globus: A metacomputing infrastructure toolkit., *Proc. of Workshop on Environments and Tools, SIAM.* (1996).

5) : Global Grid Forum. http://www.gridforum.org.