OmniRPC User's Guide Version 2.0.1

http://www.omni.hpcc.jp/omnirpc/

`<omnirpc@omni.hpcc.jp>`

Yoshihiro Nakajima
High Performance Computing Laboratory, University of Tsukuba

Mitsuhisa Sato
High Performance Computing Laboratory, University of Tsukuba

OmniRPC User's Guide Version 2.0.1: http://www.omni.hpcc.jp/omnirpc/ `<omnirpc@omni.hpcc.jp>`
by Yoshihiro Nakajima and Mitsuhisa Sato

Revision History

Revision 2.0.1 20 Dec 2007 Revised by: ynaka
New release

Table of Contents

Chapter 1. Overview of OmniRPC System

OmniRPC is a Grid RPC system which enables seamless parallel programming in cluster and Grid environment.

OmniRPC's characteristics are as follows.

- Supports typical master/worker grid applications such as parametric execution programs.
- OmniRPC provides an automatic-initializable remote module to send and store data to a remote executable invoked in the remote host. Since it may accept several requests for subsequent calls by keeping the connection alive, the data set by the initialization is re-used, resulting in efficient execution by reducing the amount of communication.
- OmniRPC inherits its API from Ninf, and the programmer can use OpenMP for easy-to-use parallel programming because the API is designed to be thread-safe. And OmniRPC supports the persistence feature, which holds the remote executable's state, and makes, programs more effective.
- Using the asynchronous call API, we can parallize a programs calling RPCs. Additionally, OmniRPC is designed to be thread-safe for easy parallel programming. We can easily parallelize existing sequential programs with direct based parallel programming such as OpenMP.
- The OmniRPC system supports local environments with **rsh**, grid environments with Globus, and remote hosts with **ssh**. Furthermore, the user can use the same program over OmniRPC for both clusters and grids because a typical grid resource is regarded simply as a cluster of clusters distributed geo-graphically.
- For a cluster over a private network, an agent process running the server host functions as a proxy to relay communications between the client and remote executables by multiplexing the communications into one connection to the client. This feature allows a single client to use approximately one-thousand remote computing hosts.
- Support of the job scheduler which considers the administrative policy of each computer resource. Now OmniRPC supports PBS and SGE as job schedulers.

Chapter 2. Installation

Requirements

In order to successfully compile and use OmniRPC system, you need the following programs and libraries which are available on most platforms as distribution packages and thereby can be installed easily.

Required:

- gcc 2.95.4 (or compatible), available at \l\l (http://www.gnu.org)
- GNU make (or compatible), available at \l\l (http://www.gnu.org)
- Remote Shell (or compatible)
- flex 2.5.4,

Optional:

- OpenSSH SSH client (or compatible), avilable at \l\l (http://www.openssh.com)
- Java SDK 1.4.0 (or higher), available at \l\l (http://java.sun.com)
- Globus Toolkit 2.4 (or compatible), avilable at \l\l (http://globus.org)

OmniRPC system was tested with RedHat Linux 7.3 on Intel Xeon 2.4, 512MB RAM and Debian GNU/Linux 2.0 (and higher version) on Intel compatible PCs.

Compilation and Installation

Compilation and Installation form Source File

In order to compile and install OmniRPC system, type the following in the base direcotry of the OmniRPC distribution:

```
% ./configure
% make
(as root)
# make install
```

As a result, the OmniRPC software is installed on the default directory ("/usr/local/omrpc"). If your system's make-command is gmake, type gmake instead of make.

If you want to change the install directory, you can specify another install directory.

```
% ./configure --prefix /path/to/install
```

After installation, add "*/path/to/install/bin*" to the PATH environmental value.

Compilation and Installation with the Globus Toolkit library

To compile OmniRPC with Globus Toolkit, the **globus-makefile-header** program, which is attached to Globus Toolkit, is needed. And, OmniRPC requires a library which supports pthread (such as gcc32dbgpthr, gcc32pthr). Type the following to build OmniRPC system with Globus Toolkit library.

```
% ./configure --enable-globus
```

At this time, the Globus Toolkit path is set to the value of ${GLOBUS_LOCATION}, and the path to Grid Packaging Toolkit(GPT) is set to the value of ${GPT_LOCATION}.

If you want to specify a path to Globus Toolkit or to GPT, you should type extra options.

```
 % ./configure --enable-globus --with-globusDir=/path/to/globus --with-gtpDir=/
```

Install Test

To test the installation, proceed as follows. Before testing, you should move "*OmniRPC/test*" directory. And, in each test, you should move the directories s which are written by number. To obtain more detailed information of the test program, especially if you encounter trouble with program testing, add the command option "`--debug`", if some trouble on program testing.

stub test (worker program test)

To generate the stub program and test the *"local-exec-test"* program, move the subdirectory (omrpc_stub_test) and type in **"make"** to compile the program.

```
% make
```

To test the stub program

```
% local-exec-test
```

Next, we should register the stub information. In registration, we have to create the directory (*`${HOME}/.omrpc_registry`*).

```
% mkdir $HOME/.omrpc_registry
% make reg_stub
```

exec-test

We test stub activation on a remote node. Especially, we test APIs (OmniRpcExecRemote, OmniRpcExecCall) which enable remote execution. First, we input **"make"** to compile the program.

```
% make
```

To test the *OmniRpcExecRemote* API, <stub_directory> is a stub directory on which we compile procedure 1. We must describe <stub_directory> as an absolute path.

```
% remote_exec_test1 <host> <stub_directory>
```

In a similar manner, test OmniRpcExecCall API.

```
% remote_exec_test2 <host> <stub_directory>
```

call-test

You should check Remote Procedure Call's API(OmniRpcCall, OmniRpcCallByHandle). Create the host file(`hostfile.xml`). You can get more detailed information about creating `hostfile.xml` from hostfile.txt.

```
% cat ~/.omrpc_registry/hosts.xml
<OmniRpcConfig>
<Host name="localhost"/>
</OmniRpcConfig>
```

Check OmniRpcCall API.

```
% rcp-call-test1
```

Check OmniRpcCallByHandle API.

```
% rpc-call-test2 <host>
```

Check OmniRpcModuleInit, OmniRpcCall API.

```
% rpc-call-test3
```

Check OmniRpcAsync, OmniRpcWaitAll API.

```
% rpc-async-test1
```

The install test is finished.

Chapter 3. Getting Started: Programming with OmniRPC

Simple example

Let's begin programming with OmniRPC using a simple example. In this case, we consider a program to calculate from 1 to 10 using sine function.

```
#include <stdio.h>
#include <math.h>

int main(){
  int i;
  double x, res[10];
  x = 0.0;
  for(i = 0; i < 10; i++){
    res[i] = sin(x);
    x += 1.0;
  }
  for(i = 0; i < 10; i++)
    printf("sin(%d)=%g\n",i,res[i]);
}
```

On this program, we calculate the sine in a remote node using OmniRPC. We label the computer node on which the program makes the Remote Procedure Call (RPC) as the *client host*, and the computer node on which procedures are executed by the RPC call as *the remote host*.

Execution environment

We assume the environment below as a simple explanation.

- Client host's name is jones.
- Remote host's name is dennis.
- From jones to dennis, we can execute "**rsh**" with no password authentication. In other words, *jones* is registered to "**/etc/host.equiv**" or the user's "**.rhost**" on the remote host.
- On both hosts, OmniRPC is installed to the default path("**/usr/local/omrpc/**").

In addition, OmniRPC does not need to share the file system in both hosts.

OmniRPC agent and remote executable programs

In OmniRPC, to activate a remote executable program on a remote node which is described in the hostfile, the omrpc-agent is first activated when OmniRpcInit, which is an initialization API is called.

This agent is activated for each program, and runs during program execution. This agent is invoked by an authentication method such as rsh, Globus GRAM, or ssh. And agent provides access to the module information which was registered on the remote node and interface of the job scheduler, communication multiplexing, and so on. If you want to know more details about these features, please see the section, *12. Hostfile Description*.

In the upper figure, agent stands for *OmniRPC-agent*, *rex* stands for remote executable program. In this example, the agent and rex are executed on the same host (dennis).

Write remote executable program

Here, we'll write a program which calculates the sine on a remote host. We will define the interface of sine function. We name the file which defines interfaces *IDL(Interface Description Language) file*. The IDL is discussed in detail later section. For example, we can write the `calc_sin.idl` file as follows.

```
Module calc_sin;

Globals {
#include <math.h>;
}

Define calc_sin(double IN x, double OUT result[]){
  *result = sin(x);
}
```

Module: We define the module name in the IDL file. In this example, we set this module name as "`calc_sin`".

Define: Interfaces are defined by the "*Define*" directive. In this example, we use sin. Arguments are the specified data type and whether the data is input or output. In OmniRPC, we cannot get the return value of the function as a value of the function's value like a original sin function. So, we get the value by specifying arguments as "*OUT*", as shown in the example. In the part surrounded with {...}, we can write a procedure which is executed on the remote host in C Language. This program calls the sine function and "*result*", which is an "*OUT*" argument, returns the since value.

Globals: In the part assigned to *"Globals"*, we can describe any C program which is necessary for functions that are defined in the modules. In this example, the IDL includes files which are required to call the "`sin`" function.

We can generate a remote executable program from the IDL file by using *omrpc-cc*, which is OmniRPC's remote executable module generator. So, let's convert using this command. ("`-lm`" option is to link math library.)

```
        % omrpc-cc calc_sin.idl -lm
```

"`calc_sin.rex`" is generated by executing this program. This is the remote executable program.

By the way, the IDL can define multiple functions in a remote executable module. For Further details, see 13. IDL file Description.

Registration of remote executable program

After making the remote executable program, you should register it. For registration, we use the "omrpc-register" program.

```
% omrpc-register -register calc_sin.rex
```

This program creates the ".omrpc_register" directory in the user's home directory and database, which consists of the module name, function name and path to the remote executable program.

After completing the above procedure, setup on the remote host side is finished. We move next to the setup on the client host side.

Client program

We rewrite the client program with OmniRpcCall.

```
#include <OmniRpc.h>;
#include <stdio.h>;
#include <math.h>;

int main(int argc,char *argv[]){
   int i;
   double x, res[10];

   OmniRpcInit(&argc,&argv);

   x = 0.0;
   for(i = 0; i < 10; i++){
       OmniRpcCall("calc_sin",x,; res[i]);
       x += 1.0;
   }
   for(i = 0; i < 10; i++)
      printf("sin(%d)=%g\n",i,res[i]);

  OmniRpcFinalize();

}
```

First, we call OmniRpcInit() before we use the OmniRPC library. And, at end of this program, we call OmniRpcFinalize(). Prototype definitions of these functions are described in OmniRpc.h, so the program must include the "*OmniRpc.h*" header file.

We compile this program.

```
% omrpc-cc -o test.exe test.c
```

You can compile without omrpc-cc when you specify the directory of OmniRPC library.

```
% cc -I/usr/local/omrpc/include \
-o test.exe test.c -L/usr/local/omrpc/lib -lomrpc_client -lomrpc_io
```

By default, OmniRPC software is installed at `/usr/local/omrpc`. If you install the software in an other directory, you should change `/usr/loca/omrpc` to the correct directory in which OmniRPC is installed.

Create hostfile.xml

In OmniRPC, usually we describe the execution environment in *hostfile.xml* with XML notation. In this example case, we describe the execution environment as follows.

```
<?xml version="1.0" ?>;
<OmniRpcConfig>;
   <Host name="prost" />;
</OmniRpcConfig>;
```

We write the above information in the *hosts.xml* file.

Execution of Cleint Program

The setup is finished. What we have to do next is to execute the program.

```
% test.exe --hostfile hosts.xml
```

This program searches for a remote function named sin on a remote host which is specified in the hosts.xml file and activates the remote module. If the "`--hostfile`" option is omitted, "`--${HOME}/.omrpc-registry/hosts.xml`" file is used.

Summary

Below list is a summary of the general procedure when using OmniRPC system.

1. Install OmniRPC on the client host and remote host.

2. On the remote host, create a remote executable program of the remote function and register it.
   a. Create the IDL file which defines interfaces.
   b. Generate a remote executable module from the IDL file with the "omrpc-cc" program.
   c. Register with omrpc-register.

3. On the client host, create hosts.xml which describes the remote host.
4. Write the client program, and compile with omrpc-cc.
5. Execute the client program, specifying hosts.xml.

Chapter 4. Parallel Programming with OmniRPC

One aim of OmniRPC is to perform parallel programming with RPC. We execute the "calc_sin" function in parallel on multiple remote hosts by showing one example of parallel programming, as in 3. Programming with OmniRPC.

Execution Environment

We assume an execution environment like the following.

- The client host's name is jones
- We use three hosts (dennis, alice, jack) as remote hosts.
- **rsh** can be executed from jones to the remote hosts without password authentication. In other words, jones is registered at */etc/host.equiv* or at the user's *.rhost* on a remote host.
- OmniRPC is installed in the default path(/usr/local/omrpc/) on both hosts.
- And, calc_sin.rex which is the remote executable program of the "`calc_sin`" function, is registered on each remote host, as in previous example.

Parallel programming with asynchronous call

We show an example of an asynchronous call with OmniRPC as follows.

```
#include <OmniRpc.h>;
#include <stdio.h>;
#include <math.h>;

int main(int argc,char *argv[]){
   int i;
   double x, res[10];
   OmniRpcRequest req[10];

   OmniRpcInit(&argc,&argv);

   x = 0.0;
   for(i = 0; i < 10; i++){
       req[i] = OmniRpcCallAsync("calc_sin",x,&res[i]);
       x += 1.0;
   }
   OmniRpcWaitAll(10,req);
   for(i = 0; i < 10; i++)
      printf("sin(%d)=%g\n",i,res[i]);

  OmniRpcFinalize();

}
```

The OmniRpcCallAsync API activates the remote procedure call and returns without waiting for its call termination. Its API returns an OmniRpcRequest value which corresponds to the remote procedure call as a return value. We store its value in an array, and by using OmniRpcWaitAll API, the program waits for termination of all RPC calls. For more details about this API, see 15. C API index.

Hostfile and Execution

Before program execution, you should prepare a hostfile.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="alice" />
   <Host name="dennis" />
   <Host name="jack" />
</OmniRpcConfig>
```

The rest of the procedure is the same the former example. The relationship between the agent and rex is shown below.

Parallel programming with OpenMP

We can do parallel programming by calling OmniRpcCall on a multi-thread program with Omni OpenMP (http://phase.hpcc.jp/Omni/), which is one of OpenMP compiler of an implementation using a thread. We show an example as follows.

```
#include <OmniRpc.h>
#include <stdio.h>
#include <math.h>

int main(int argc,char *argv[]){
   int i;
   double x, res[10];
   OmniRpcRequest req[10];

   OmniRpcInit(&argc,&argv);

   x = 0.0;
#pragma omp parallel for
   for(i = 0; i <; 10; i++){
       req[i] = OmniRpcCall("calc_sin",x,&res[i]);
       x += 1.0;
   }
   for(i = 0; i < 10; i++)
      printf("sin(%d)=%g\n",i,res[i]);

  OmniRpcFinalize();

}
```

When you want to run this program, remember to set the OMP_NUM_THREADS environmental value, which specifies the number of OpenMP threads, to a value more than the number of remote hosts.

Chapter 5. Use of shared memory multiprocessor (SMP) system

The shared memory multiprocessor (SMP) system is a parallel computer which has multiple processors. High-end systems and, recently, some PCs haves multiprocessors. On a remote host, a program will show good performance when the program uses these CPUs at the same time.

Setting for SMP

In this example, we assume the environment below.

- Client host is jones.
- Remote host(apple) is an SMP system and has 4 processors.
- The same network and process is invoked by **rsh**.

Setting of Maxjob

Multiple remote execution programs can be executed at the same time on each processor in an SMP system. In this case, we can execute a maximum of 4 programs. We describe hostfile for the SMP system as follow.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="apple" arch="i386" os="linux">
   <JobScheduler maxjob="4" />
   </Host>
</OmniRpcConfig>
```

In this description, we set the maxjob attribute as "4" for the JobScheduler element. This shows that the maximum number of remote executable programs executed on the remote host at the same time is 4. Certainly, in a 4-way SMP system, we can run more than 4 jobs, but in practice 4 processes are executed in parallel. Therefore effectiveness may not change.

In this case, the agent invoke multiple rex-es. The relationship is as follows.

Execution of client progra

The client program is executed in the same manner as the others, except for using the above host file.

```
% a.out --hostfile hosts.xml  args
...
```

Of course, to achieve good performance, we use a client program which uses the OmniRPC call in parallel.

Chapter 6. Execution in Globus Toolkit Environments

So far, we've mentioned local environments which can use rsh, but rsh cannot be used in wide-area network environments. One of the solution for that is the Globus toolkit.

We explain the case of using Globus Toolkit, which is the de facto standard for constructing grid environments. If you use Globus Toolkit, on a remote host, creating the remote executable program, registering and programing is the same as previously stated. The only detail which is different is describing the hostfile. If you don't know much about the Globus Toolkit, please see Globus Alliance (htttp://www.globus.org).

Execution Environment with Globus Toolkit

We assume an execution environment as follows.

- Client host is *alice.hpcs.is.tukuba.ac.jp* .
- Remote host is *dennis.hpcc.jp*. The Globus-gate-keeper must be running on the remote host.
- On the remote host, a non special privilege port (the port number is more than 1024) may be opened.

Warning

Please refer to the Globus manual for the method to limit the range of Globus ports.

In the explanation below, we assume an environment setting on which a job can be submitted from the client host to the remote host.

Preparation of Globus

First, initialize the proxy certificate when you use Globus Toolkit.

```
% grid-proxy-init
```

In this phase, you should input the pass phase, and create the proxy certificate. You can check whether Globus can execute normally or not, by the example.

```
% globusrun -o -r dennis.hpcc.jp '& (executable=/bin/date)'
```

If you cannot see the current time, you have to return to the setting of Globus. On other hand, if you can see the current time, move to the next step.

Hostfile for Globus

To use *dennis.hpcc.jp*, you should describe the information inn hosts.xml, and you can execute the client program by its hostfile. When you use Globus, you should describe the following information.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="dennis.hpcc.jp" arch="i386" os="linux">
   <Agent invoker="globus" />
   </Host>
</OmniRpcConfig>
```

This description is for an agent using Globus to invoke a remote executable program. In this setting, the relationship between the agent and rex is shown in the following.

Execution of client program

You can execute the client program in the same manner without using the hosts.xml file.

```
% a.out --hostfile hosts.xml  args ...
```

Chapter 7. Execution of ssh Environment

SSH (secure shell) is a common method for using remote machines. In this section, we explain execution of OmniRPC applications using ssh. In addition, like using Globus, create a remote executable module, register a program, and to create program are almost the same. The only different details is the description of hostfile (hosts.xml).

Preparing for SSH

In this example, the environment is described below.

- Client host is `alice.hpcs.is.tukuba.ac.jp` .
- Remote host is `dennis.hpcc.jp`.
- There is not a firewall between the remote host and client host. In other words, programs can communicate without limitation on the non special privilege port.

We assume that we can use SSH here. That is, we assume that we can access with SSH from `alice.hpcs.is.tsukuba.ac.jp` to `dennis.hpcc.jp`.

When using SSH, you should set up auto authentication with an **ssh-agent**. If you do not, you have to type in the password at each remote host. If you want to know more detail, see man of "**ssh-agent**", In this example, we outline the usage below.

1. Activate **ssh-agent**, and set the environment variable on your terminal.

   ```
   % eval `ssh-agent`
   ```

2. Register the pass phrase with **ssh-add**.

   ```
   % ssh-add
   type your pass phrase here.[passwd]
   Identity added: /home/foo/.ssh/id_rsa (/home/foo/.ssh/id_rsa)
   % ssh-add
   ```

With the above procedure, you should confirm whether auto authentication is running or not.

```
%  ssh dennis.hpcc.jp
```

With the above command, you can login to remote hosts without typing the password.

Hostfile for SSH

To use `dennis.hpcc.jp`, you should describe **hosts.xml**, and execute the client program with this hostfile. Write the hostfile for using ssh, as follows.

```
<?xml version="1.0" ?>
```

```
<OmniRpcConfig>
   <Host name="dennis.hpcc.jp" arch="i386" os="linux">
   <Agent invoker="ssh" />
   </Host>
</OmniRpcConfig>
```

This description is for activating the agent with ssh. In this specification, the relationship between the agent and rex is shown below.

Setting for firewall: Using SSH's port forwarding and MXIO option.

There will be some firewalls between remote hosts and the client host when the client host and the remote hosts extend across administrations. When we use ssh for agent activation, programs communicate with the client and agent, using *ssh port forwarding* . Using the ssh port forwarding function, we can communicate between remote executable modules on the remote host and client program. This is OmniRPC's multiplex communication function. If you want to use this feature, specify the "*mxio*" attribution in the agent.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="dennis.hpcc.jp" arch="i386" os="linux">
   <Agent invoker="ssh"  mxio="on"  />
   </Host>
</OmniRpcConfig>
```

The relationship of the agent and rex with this option is as follows. In this case, communication with rex and the client program occurs by way of rex.

Execution of client program

You can execute in same manner without using hosts.xml.

```
% a.out --hostfile hosts.xml  args...
```

Of course, to achieve good performance, we use a client program which uses the OmniRPC call in parallel.

Setting of Login name

In addition, if the user name is different on client host and remote host, describe hostfile.xml as follows.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="dennis.hpcc.jp" user="foo" arch="i386" os="linux">
   <Agent invoker="ssh" mxio="on"  />
   </Host>
</OmniRpcConfig>
```

Chapter 8. Execution on Clusters

On grid environments, clusters which connects many PCs and workstations in a network is a typical computation resources. In OmniRPC, we can treat a cluster as a remote host. We can run a remote executable module on each node in the cluster and execute in parallel so that we can achieve good performance.

Setting of cluster environment

When using clusters, we can access at least one computer in a cluster form the client host. We label this computer as the cluster server host and label the computers in the cluster without a cluster server host as the cluster node host.

- Client host is *`jones.tsukuba.ac.jp`* .
- Cluster server host is *`hpc-serv.hpcc.jp`*. So *`hpc1, hpc2 and hpc3`* are connected to the cluster server host as cluster node hosts.
- Both the cluster server host and cluster node hosts share the same file system.
- Client host can connect directly to the cluster server host and all of the cluster nodes. All port are not limited.

The last item in the list assumes that the client host and cluster are in same network.

Selection of job scheduler

In OmniRPC, the omrpc-agent is invoked first on the cluster server host, and this agent activates remote executable module on each cluster node host with the appropriate scheduler. We can use one of the scheduler described below.

- Built-in round-robin scheduler (rr)
- Portable Batch System (PBS)
- SunGridEngine (SGE)

Use of built-in round-robin scheduler

OmniRPC's built-in round-robin scheduler is a simple scheduler which is implemented in the agent. This scheduler activates remote executable modules on cluster node hosts usign **rsh**.

To use this scheduler, we create a nodes file which specifies cluster node hosts on the registry (*`"$HOME/.omrpc-register"`*) of the cluster server host. Below is the setting for this example.

```
hpc1
hpc2
```

```
hpc3
```

If you want to use **ssh** instead of **rsh** inside of a cluster, please add *ssh* to next of hostname like below file.

```
hpc1 ssh
hpc2 ssh
hpc3 ssh
```

On the client host side, we create this hostfile.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="hpc-serv.hpcc.jp" arch="i386" os="linux">
   <JobScheduler type="rr" maxjob="4" />
   </Host>
</OmniRpcConfig>
```

Set the type attribute in the job scheduler element to the round-robin scheduler "*rr*." The default value for this attribute is "*fork*," which just creates the process on the same host. Our example applies to an SMP system. The number of cluster node hosts is 4, so you should set maxjob equal to 4.

The relationship between the agent and rex with this option is as follows.

Cluster in private network

In the above example, the client host and cluster hosts are in same network. Also, the remote executable programs which execute on the cluster node hosts are activated directly for the client host. In the case in which the cluster and client host are in different networks, programs can communicate to the client host from a cluster node host.

But, as the number of node hosts increases, so do the clusters connected to the local-address network. In this situation, only the server host has a global IP address; the node hosts have local IP addresses. For OmniRPC, the cluster node host must communicate with the client host, but in this situation the cluster node host cannot communicate directly with the client host outside the cluster's network.

In this situation, there are 2 ways to use the cluster.

1. Set NAT to communicate with outside networks from the cluster node hosts. Programs can connect to anonymous ports on the client node from each cluster node. For the setting of NAT, please refer to NAT documents.
2. By using the agent function of multiplex communication, the agent relays communications between remote executable programs, which are executed on the cluster node host and client host.

We show an example hostile.xml which is based on the second way to use the cluster.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="hpc-serv.hpcc.jp" arch="i386" os="linux">
   <Agent invoker="rsh" mxio="on" />
   <JobScheduler type="rr" maxjob="4" />
   </Host>
</OmniRpcConfig>
```

You should set the *mxio* attribute on the agent element with "*on*." In this case, because we assume that the cluster server host and client host are in same network, we use "**rsh**." If you want to invoke the agent with SSH, set "**ssh**". If you want to do this with the Globus gate keeper, use "**globus**."

Using this option, the relationship is shown in the figure below.

The agent relays communications between every rex which is executed on the remote node host and the client.

Cluster outside firewall

You don't have to prepare for this situation.

Cluster inside firewall

We now explain the case of using clusters from outside of firewalls. When there are firewall(s), it is necessary at least to access with ssh to the cluster server host. If you can not use anonymous ports without a port (#22) of ssh, you can use the function of multiplex communication with the agent. We show the hostfile for this example.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="hpc-serv.hpcc.jp" arch="i386" os="linux">
   <Agent invoker="ssh" mxio="on" />
   <JobScheduler type="rr" maxjob="4" />
   </Host>
</OmniRpcConfig>
```

Set the *mxio* attribute "*on*" in the agent element and use the function of multiplex communication.

In environments which use Globus Toolkit, usually there are no firewalls, so you don't have to prepare. But, you have to set the mxio attribute in the same manner if the clusters consist of private IP addresses.

The relationship between the agent and rex is shown by the figure below.

Communications between the client and **rex**, which are executed on remote node hosts are relayed by the agent. And communications between the agent and clients are relayed by SSH's port forwarding through the firewall.

Chapter 9. Programming with OmniRpcHandle API

OmniRpcHandle is a data structure used for connection with a specific remote executable program. Using OmniRpcHandle , you can write programs which keep the status on a remote executable program. And, you can allocate a remote executable program on a specific remote host.

What's OmniRpcHandle

OmniRpcHandle is a data structure which presents a connection with a specific remote executable program. OmniRpcHandle is created by activating the remote executable program which corresponds to the module with the OmniRpcCreateHandle API. Once the remote executable program is executing, remote executable programs can accept the requests of another RPC call which is inside of same module, and the program does not need to exit after finishing calculation using a function in the module.

Using this feature, you can do the following.

- Keep the status on the remote executable program side. For instance, in function B you can use data which are set by function A in the same module. In other words, you can reuse the data sent to modules.
- Specify a remote node on which remote executable modules are activated.

With OmniRpcCall API, you specify the remote function only, and call the RPCs. From function name, the client program searches for the modules which contain it. Also, on adequate remote hosts remote program can run the remote executable program which corresponds to it, and assign. However if the client program uses OmniRpcHandle API, you should program the host executable module on which the programs are allocated. The host executable module should be allocated on a remote host. However, problems may arise when remote executable modules fail or are unavailable due to a timeout.

Programming with OmniRpcHandle

For an easy example, we show a program which adds values which set the in setAB function and get its value, and use the IDL file as is.

```
Define Sample;

Globals {
 int a,b;
}

Define setAB(int in a0, int in b0)
{
   a = a0; b = b0;
}
```

```
Define plusAB(int out ab[])
{
     *ab = a + b;
}
```

In the Globals directive, we define, in a style similar to C language, the variables which are used in the whole module. In the Globals scope, we write the module definitions in the style of C language.

You should compile this module with **omrpc-cc**, and register **Sample.rex** on the remote host (*`alice.is.tsukuba.ac.jp)`*) with **omrpc-register**.

```
#include <OmniRpc.h>
#include <stdio.h>

int main(int argc,char *argv[]){
   int ab;
   OmniRpcHandle handle;

   OmniRpcInit(&argc,&argv);
   handle = OmniRpcCreateHandle("alice.is.tsukuba.ac.jp","Sample");

   OmniRpcCallbyHandle(handle,"setAB",10,20);
   OmniRpcCallbyHandle(handle,"plusAB",&ab);
   printf("a+b=%d\n",ab);

   OmniRpcHandleDestroy(handle);

   OmniRpcFinalize();
   exit(0);
}
```

After the modules are initialized, the client program first activates the remote executable programs with OmniRpcCreateHandle API, and gets the OmniRpcHandle corresponding to it. Otherwise, by using OmniRpcCallbyHandle, the client program can call functions in the modules. Finally, the client program can stop the remote executable program with OmniRpcDestroyHandle API.

Also available is OmniRpcCallAsyncByHandle API, which call RPCs asynchronously.

Acquisition of host information.

With OmniRpcCreateHandle API, you specify the remote host name on which the remote executable modules run. However, if you don't, programs allocate the appropriate remote host on modules which are registered.

Chapter 10. Automatic module initialization with OmniRpcModuleInit API

By using OmniRpcModuleInit API, when modules of a remote executable program are activated, the call initialize function is enabled automatically. So, you can write programs efficiently for master/worker programs which require worker initialization. We call this OmniRPC's restricted persistency model "Automatic Initializable Module".

What's auto module initialization

Auto module initialization is a function which calls the initialization method automatically when modules of a remote executable program are invoked.

By using OmniRpcHandle API, you can write efficient programs which keep data on a remote executable program, but you should write the program which is used by the remote executable program. We only specify the remote functions which are specified by OmniRpcCall at the start, the OmniRPC system executes and runs appropriate remote executable programs for requests. But we cannot know which remote executable programs are allocated by the OmniRPC system, so it is impossible to set the data beforehand.

In some OmniRPC master/worker programs, the master and worker share common data. Workers calculate its data with different parameters which are taken from the master. Auto module initialization is convenient in cases like this. In OmniRPC, if there is a function named "Initialize" in a module, "Initialize" is called automatically when a new remote executable program is activated for an other function's RPC call. Common data are set in initialization, and it can be efficient to reuse this data when function calls are called by real varied parameters. The bigger the costs of setup, the bigger are the effects.

Programming with OmniRpcModuleInit API

For example, we will use a program to calculate the appearance of 10 sorts of strings (at a maximum of 10 characters) from a string which is 10000 characters in size. The sequential version may be written as follows.

```
#include <stdio.h>
#include <string.h>

char data[10000];  /* data to be searched */
char str[10][10];  /* string to be compared */
int occurrence[10];  /* array to record occurrence */

/* prototype */
void count_occurrence(char *data,char *str, int *r);

int main(int argc, char *argv[])
{
    FILE *fp;
    int i;

    if((fp = fopen("data","r")) == NULL){
fprintf(stderr,"cannot open data file\n");
exit(1);
    }
    fread(data,10000,1,fp);
    fclose(fp);

    if((fp = fopen("strings","r")) == NULL){
fprintf(stderr,"cannot open strings file\n");
exit(1);
    }
    for(i = 0; i < 10; i++)
fscanf(fp,"%s",str[i]);
    fclose(fp);

    for(i = 0; i < 10; i++)
count_occurrence(data,str[i],&occurrence[i]);

    for(i = 0; i < 10; i++)
printf("string(%i,'%s') occurrence=%d\n",i,
       str[i],occurrence[i]);

    exit(0);
}

void count_occurrence(char *data,char *str, int *r)
{
    int i,len,count;
    len = strlen(str);
    count = 0;
    for(i = 0; i < 10000-len; i++){
if(strncmp(&data[i],str,len) == 0) count++;
    }
    *r = count;
}
```

To parallize this program with OmniRpcCallAsync, we calculate count_occurrence in the remote executable program. The IDL file may be like the example shown below.

```
Module count_occurrence;

Define count_occurrence(char IN data[10000],char IN str[10], int OUT
r[]) Calls "C" count_occurrence(data,str,r);
```

Link this with the count_occurrence function. and register it. In the client program, we change a call of count_occurrence to a call of OmniRpcCallAsync API.

```
int main(int argc, char *argv[])
{
    FILE *fp;
    int i;
    OmniRpcReqeust reqs[10];

    OmniRpcInit(&argc,&argv);

    ... /* input data */

    for(i = 0; i < 10; i++)
     reqs[i] = OmniRpcCallAsync("count_occurrence",
                      data,str[i],&occurrence[i]);
   OmniRpcWaitAll(10,reqs);

    ...
   OmniRpcFinalize();
   exit(0);
}
```

In this case, there is the problem that data are sent for each RPC call on the remote executable module. This data is unchanged, so it is efficient to reuse the data sent on the remote executable program side.

By using OmniRpcModuleInit API, the client program sends the data in the initialization of the remote executable programs, and sends only the strings which are searched with OmniRpcCall. We define the IDL file as follows.

```
Module count_occurrence;

Globals {
#include <string.h>
char data[10000];
}

Define Initialize(char IN input_data[10000])
{
     memcpy(data,input_data,10000);
}

Define count_occurrence_each(char IN str[10], int OUT r[]){
    count_occurrence(data,str,r);
}
```

In the client program, initialization is described.

```
int main(int argc, char *argv[])
{
    FILE *fp;
    int i;
    OmniRpcReqeust reqs[10];

    OmniRpcInit(&argc,&argv);

    ... /* input data */

    OmniRpcModuleInit("count_occurrence",data);
    for(i = 0; i < 10; i++)
     reqs[i] = OmniRpcCallAsync("count_occurrence_each",
                        str[i],&occurrence[i]);
   OmniRpcWaitAll(10,reqs);
    ...
   OmniRpcFinalize();
   exit(0);
}
```

We specify the necessary data in the initialization with OmniRpcModuleInit. The initializations are indeed taken when the necessary remote executable modules are activated. So, it is important not to write in the area addressed by the pointer variables in initialization.

Chapter 11. Direct Execution of Remote Program

Usually, in OmniRPC we execute remote executable programs via the agent. It is possible to execute directly to specify a remote executable module.

The case of direct activation of remote executable program

Usually in OmniRPC, the client program executes a remote executable program via the agent. Here are some advantages.

- By registering to the registry, it is possible to activate remote executable modules on the remote host without knowing its path. Furthermore, in the case of the same module's name, it is possible to allocate the job to the appropriate remote host.
- When the remote host is a cluster, it is possible to allocate jobs to the cluster node host.

In this explanation, without the agent, we introduce a method to execute directly the remote executable module and to call RPCs. If there are remote executable programs for which you know the path in a certain remote host, you can omit registering remote executable programs, and so on. The advantages as are listed below.

- Without describing a communication protocol, programs call function by using OmniRPC protocol.
- By activating using Globus and ssh, the function of port forwarding is available in authentication.

Setting environment for remote nodes.

- Client host is *`jones.tsukuba.ac.jp`*.
- Remote host is *`dennis.hpcc.jp`*.
- Set **calc_sin.rex**, which we introduced already as an example program, to "*`/usr/local/tmp/`*" .
- Globus Toolkit's gate-keeper is running on *`dennis.hpcc.jp`*, and it is possible to execute programs with GRAM.

If you use this function, it executes rex directly without the agent.

API for direct execution of remote program

Because, we don't use the agent for direct execution, initialization of the library is taken by OmniRpcExecInit() API.

APIs which activate remote executable programs on the remote host, are like APIs which use OmniRpcHandle. OmniRpcExecOnHost() API enables the execution of a remote executable program, which is specified by its path, on the specified remote host, and it returns OmniRpcExecHandle, which presents its connection. By using OmniRpcExecCall it is possible to call a function which is inside a module.

We show this example below.

```
#include <OmniRpc.h>
#include <stdio.h>

int main(int argc,char *argv[]){
   double r;
   OmniRpcExecHandle handle;

   OmniRpcExecInit(&argc,&argv);
   handle = OmniRpcExecOnHost("dennis.hpcc.jp","/usr/local/tmp/calc_sin.rex");

   OmniRpcExecCall(handle,"calc_sin",10,&r);
   printf("sin(10)=%g\n",r);

   OmniRpcExecTerminate(handle);

   OmniRpcExecFinalize();
   exit(0);
}
```

OmniRpcExecTerminate API enables the termination of the remote executable program which responds to the handle. Finally, you should use OmniExecFinalize() at the end of the program.

Program execution

You should specify the path of an executable program to activate the remote executable program. For example, the case of an execution using Globus is as follows.

```
% a.out --globus args...
```

In the case of SSH activation, you specify "`--ssh`."; if don't specify this, rsh is used to activate.

The limitation of direct activation of remote executable programs.

At this time, there are some limitations with this method.

- No support for a multi-thread environment (no thread-safe)
- No support for asynchronous calls.
- No support for activating with multiple methods of activation.

We hope to improve these issues in the future.

Chapter 12. File Transfer in OmniRPC

File Transfer using Filename

OmniRPC supports file transfer between client program and remote executable program. In file transfer mode, you should spcify filenames which you want to transfer files.

Write Remote Executable Program

For an easy example, we show a program which concatinates 2 files and use the IDL file as is.

```
Module testfile;

Globals {
#include <math.h>
}

Define testfile(IN filename infile[2], OUT filename outfile[])
{
    FILE *infp, *outfp;
    char tmp[128];
    int i;

    fprintf(stderr, "STUB : infile[0] : %s\n", infile[0]);
    fprintf(stderr, "STUB : infile[1] : %s\n", infile[1]);


    if((outfp = fopen(*outfile, "w+")) == NULL){
        perror("fopen");
        exit(1);
    }
    for(i = 0; i < 2; i++){
        if((infp = fopen(infile[i], "r+")) == NULL){
            perror("fopen");
            exit(1);
        }

        while(fgets(tmp, sizeof(tmp), infp) != NULL){
            fprintf(outfp, "%s", tmp);
        }
        fclose(infp);
    }

    fclose(outfp);
    fprintf(stderr, "STUB : END\n");
    fflush(stderr);
}
```

In this source code, special identification *filename* is introduced for file transfer. Type of *filename* is string (which is a alias of *`char *`*). In this example, *`infile[2]`* is string typed array variable. At the execution time, OmniRPC remote executable create new filename before the procedure in remote program. So you can use infile as filename strings. But output variable of outfile in IDL file is filename pointer, that alias is *`char **`*.

We can generate a remote executable program from the IDL file by using omrpc-cc. So, let's convert using this command.

```
% omrpc-cc testfile.idl
```

Write Client Program

```
#include <OmniRpc.h>
#include <stdio.h>

int main(int argc,char *argv[])
{

   char *infile[2] = {"a.txt", "b.txt"};
   char *outfile = "a.out";
   char tmp[128];
   FILE *fp;
   int i,j;
   int c = 0;

   OmniRpcInit(&argc,&argv);

   for(i = 0 ;i < 2; i++){
       if((fp = fopen(infile[i], "w+")) == NULL){
           perror("cannot open file");
           exit(1);
       }
       for(j =0; j < 128; j++)
           fprintf(fp, "%i\n",c++);
       fflush(fp);
       if(fclose(fp) != 0){
           perror("cannot close file");
       }
   }

   OmniRpcCall("testfile", infile, &outfile);

   fprintf(stderr, "TRANSFER FIN\n");
   fflush(stderr);

   if((fp = fopen(outfile, "r+")) == NULL){
       perror("cannot open file");
       exit(1);
```

```
    }
    while(fgets(tmp, sizeof(tmp), fp) != NULL){
        fprintf(stdout, "OUTPUT:%s", tmp);
    }

  OmniRpcFinalize();
}
```

We can generate a client program from the C file by using omrpc-cc. So, let's convert using this command.

```
        % omrpc-cc -o testfile testfile.c
```

Execution of Program

Before you execute this example, you should prepaer 2 file (named a.txt and b.txt) in the same directory on which client program is.

You can execute the client program in the same manner without using the hosts.xml file.

```
% testfile --hostfile hosts.xml
```

After execution, you can see the file named "a.out" in the the same direcotry of client program.

Setting of WorkingDirectory

OmniRPC remote executable uses a temporary directory to store files. Default temporary directory is "/tmp", but you can use another directory by specifying the *WorkingPath* in hostfile. And your login id has write and read permission on that directory. We show an example of hostfile which specify temporary directory.

```
<OmniRpcConfig>
   <Host name="alice.hpcc.jp" user="foo" arch="i386" os="linux">
        <Agent invoker="globus" mxio="on" path="/usr/local/omrpc"/>
        <JobScheduler type="rr" maxjob="6" />
        <Registry path="/home/foo/app/stubs" />
        <WorkingPath path="/home/foo/tmp" />
   </Host>
</OmniRpcConfig>
```

Chapter 13. Programming in FORTRAN

In this section, we explain how to program OmniRPC in FORTRAN. If you want to know more details about APIs, see Section FORTRAN API.

A simple example in FORTRAN

Let's think about this Fortran program. This program calculates the inner product of a matrix. The main program is called main.f, and ip.f is the subroutine that calculates the inner product.

```
main.f

      double precision a(10),b(10),r
      do i = 1,10
         a(i) = i
         b(i) = i+10
      end do
      call innerprod(10,a,b,r)
      write(*,*) 'result=',r
      end
```

```
ip.f

      subroutine innerprod(n,a,b,r)
      integer n
      double precision a(*),b(*),r
      integer i
      r = 0.0
      do i = 1,n
        r = r+a(i)*b(i)
      end do
      return
      end
```

Let's set this program to call the innerprod subroutine with OmniRPC.

Create remote executable program

As in a C program, we create a program which executes a subroutine on remote hosts. So, we define an interface to innerprod. We set the module name as `f_innerprod`, and the function name as `innerprod`.

```
Module f_innerprod;

Define innerprod(IN int n, IN double a[n], IN double b[n],
        OUT double result[1])
Calls "Fortran" innerprod_(n,a,b,result);
```

The argument which specifies the array size must be a scalar variable. The double precision type in FORTRAN is "`double`", the real type is "`float`". If you directly call a FORTRAN function with calls, you specify "Fortran". Calling a function can mangle the function name in FORTRAN. Usually, in the case of the FORTRAN compiler, a mangled name is a name to which "_" has been added. Please pay attention to whether a function name contains "_"; if it is mangled, add "__" in g77 (gcc).

This definition is the same as the above definition. As scalar variable is taken as the address pointer.

```
Module f_innerprod;

Define innerprod(IN int n, IN double a[n], IN double b[n],
        OUT double result[1])
{
    innerprod_(&n,a,b,result);
}
```

Generate OmniRPC's remote executable modules from this IDL file. We name this IDL file `f_ip.idl`.

```
%  omrpc-fc f_ip.idl ip.f
```

When you run this command, the remote executable program f_ip.rex is generated. To register with **omrpc-register**, the method is the same as in C language.

```
%  omrpc-register -register f_ip.rex
```

The command **omrpc-fc**, it compiles FORTRAN program with **f77**. If you want to use another compiler, use the "`-fc`" option. For example, if you want to use the intel Fortran compiler of ifc, run the following command.

```
%  omrpc-fc -fc ifc f_ip.idl ip.f
```

Client program in Fortran

And now, change the client program main.f to call the remote executable program with OmniRPC.

```
      double precision a(10),b(10),r
      call OMINRPC_INIT
      do i = 1,10
         a(i) = i
         b(i) = i+10
      end do
      call OMNIRPC_CALL("f_innerprod*",10,a,b,r)
      write(*,*) 'result=',r
      call OMNIRPC_FINALIZE
      end
```

This program initializes with `OMNIRPC_INIT` and calls by `OMNIRPC_CALL`. Please keep in mind that the entry name which is specified by `OMNIRPC_CALL` should have "*" at the end of the name which is defined on the interface. You should add "*" to the end of strings which are obtained with OmniRPC, such as the module name and host name. OmniRPC considers "*" to be a string terminator in OmniRPC's Fortran API.

Using omrpc-fc to compile this file.

```
% omrpc-fc main.f
```

An asynchronous call in FORTRAN

Finally, we show an example which is written in FORTRAN. this example is also shown in the section Parallel programming with OmniRPC .

```
      double precision res(10)
      double precision x
      integer ireqs(10)
      call omnirpc_init
      x = 0.0
      do i = 1, 10
         call omnirpc_call_async(ireqs(i),'calc_sin*',x,res(i))
         x = x + 1.0
      end do
      call omnirpc_wait_all(10,ireqs)
      do i = 1, 10
         write(*,*) 'res(',i, ')=',res(i)
      end do
      call omnirpc_finalize
      end
```

For API details, see Fortran API

Appendix A. Description of hostfile

The hostfile is an XML file that describes the execution environment. We will show the procedure to describe it.

Specifying the hostfile

In the hostfile, you should specify which host a client program uses. You should type the command option "`--hostfile`" for the client program.

```
% a.out --hostfile  host_file args...
```

If "`--hostfile`" is not set, by default the `hosts.xml` in each user's registry is used. In other words, "`$HOME/.omrpc-registry/hosts.xml`" is nnused as the default setting.

How to describe

We show an example below.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="jones.is.tsukuba.ac.jp">
   <Host name="alice.hpcc.jp" user="foo" arch="i386" os="linux">
<Agent invoker="globus" mxio="on" path="/usr/local/omrpc"/>
<JobScheduler type="rr" maxjob="6" />
<Registry path="/home/foo/app/stubs" />
        <WorkingPath path="/home/foo/tmp" />
<Description>
This is a sample host description.
</Description>
   </Host>
   <TimeOut second="20">
</OmniRpcConfig>
```

We specify 2 hosts(*`jones.is.tsukuba.ac.jp`* and *`alice.hpcc.jp`*). In *`jones.is.tsukuba.ac.jp`*, the defaults setting is used, so the invocation method of the agent is "**rsh**" and a remote executable program is allocated. Because *`alice.hpcc.jp`* is a remote server node, Globus Toolkit's GRAM is used as a agent invocation method. Registry is at "`/home/foo/app/stubs`", not the default setting. Remote executable programs are executed by the *round-robin scheduler*, which is the built-in scheduler in the OmniRPC agent. 6 remote executable programs are invoked. Also, the account in *`alice.hpcc.jp`* is "*foo* and worker programs use "`/home/foo/tmp`" directory to store temporary files, not the default setting (default directory is "`/tmp`"

Details of Hostfile

Hostfile is an XML file which has OmniRpcConfig at the top level. OmniRpcConfig's element is Host.

Host element

In the host element, you should describe the hosts which you use.

Name attribute:(Required)

You should specify the host name with the name attribute.

User attribute: (Optional, user name on client host if omitted)

You can set the user name when the user names are different on the client host and remote host. If the value of the user attribute is omitted, the user name on the client host is used.

Arch attribute, os attribute(Optional)

At this writing, you can specify the architecture (arch) and operating system (os), but these values are not used.

In the host, you can specify the attributes below.

Agent element: (Optional)

In OmniRPC, the **omrpc-agent** is invoked in initialization, and the agent element is a option for the agent. It has no elements. but you can specify the following attributes.

invoker attribute: *`rsh, ssh, gram, globus`* (Required if agent element used)

You should specify the method of the agent invocation. "gram" is an alternative name for globus. If the agent element is omitted, the default invoker is rsh.

mxio attribute: *`on, off`* (Optional, off if omitted)

You should specify whether or not you use multiplex communication. Set this value on if you want to use the relay of communication by the agent. If you don't specify the value of the mxio attribute, the default value is off.

path attribute: (Optional, `/usr/local/omrpc/` if omitted)

On this host, if OmniRpc software is not installed in the default install path (/usr/local/omrpc), you should specify the path attribute as the install path.

JobScheduler element: (Optional)

Specify the jobscheduler of the omrpc-agent which executes the remote executable. There are some attributes. If this element is omitted, the default type is fork and maxjob is 1.

type attribute: *`fork, round_robin, rr, pbs, sge`* (Required if jobschedular element used)

> You specify the type of jobscheduer. "rr" is an alternative name for "round_robin". "pbs" stands for portable batch system and sge stands for sun grid engine.

maxjob attribute: (Option. 1 if omitted)

> You specify the number of maximum jobs which can be executed on the remote host. If the value is omitted, maxjob is set to 1.

Registry element: (Optional)

Specifys the path to the registry on the remote host. If this element is omitted, the registry path is the home directory on the remote host.

path attribute: (Required if registry element used)

> You should specify the path to registry.

WorkingPath element: (Optional)

Specifys the directory on which worker program in remote side can store the file.

path attribute: (Required if registry element used)

> You should specify the path that workers store temporary files .

Description element: (Optional)

You can describe information about host.

TimeOut element: (Optional)

Specifys the timeout seconds of connection between client program and OmniRPC agent. Default seconds is 15 .

second attribute: (Required if registry element used)

> You can specify the timeout second when the client program invokes OmniRPC agent in remote nodes. Sometimes the client program fails because of long phease of authentication. If you meet that case, please increase that value.

Debug element: (Optional)

If you want OmniRPC client program to show debug message, please write Debug element in hostfile.

DTD (Document Type Definition) of hostfile.

```
<!ELEMENT OmniRpcConfiguration (Host+, TimeOut?, Debug?)>
<!ATTLIST OmniRpcConfiguration version CDATA>

<!ELEMENT Host (Agent?, JobScheduler?, Registry?, WorkingPath?, Description?)>
<!ATTLIST Host name CDATA #REQUIRED>
<!ATTLIST Host user CDATA >
<!ATTLIST Host arch CDATA>
<!ATTLIST Host os CDATA>

<!ELEMENT Agent EMPTY>
<!ATTLIST Agent invoker (rsh|ssh|globus|gram) #REQUIRED>
<!ATTLIST Agent mxio (on|off)>
<!ATTLIST Agent path CDATA>

<!ELEMENT JobScheduler EMPTY>
<!ATTLIST JobScheduler type (fork|rr|round_robin|pbs|sge) #REQUIRED>
<!ATTLIST JobScheduler maxjob CDATA>

<!ELEMENT Registry EMPTY>
<!ATTLIST Registry path CDATA #REQUIRED>

<!ELEMENT WorkingPath EMPTY>
<!ATTLIST WorkingPath path CDATA #REQUIRED>

<!ELEMENT TimeOut EMPTY>
<!ATTLIST TimeOut second CDATA #REQUIRED>

<!ELEMENT Debug EMPTY>

<!ELEMENT Description (#PCDATA)>
```

Appendix B. IDL (Interface Description Language)

To create a remote executable program, you need to write an IDL file which describes the interfaces of each remote function. In this section, we explain IDL.

IDL file and generation of remote executable programs

We define a remote executable module in the IDL file. The stub generation program (**omrpc-gen**) from the IDL file description creates a remote executable program which communications with the remote executable module. A command (**omrpc-cc**) is a driver that generates the remote executable program and , also serves as link to library.

Example

We show an example as follows.

```
Module mat_mult;

Define dmmul(mode_in int n, mode_in double A[n][n],
        mode_in double B[n][n],
        mode_out double C[n][n])
{
    double t;
    int i,j,k;
    for (i=0;i<n;i++){
        for (j=0;j<n;j++){
            t = 0;
            for (k=0;k<n;k++){
                t += A[i*n + k] * B[k*n+j];      /* inner product */
            }
            C[i*n+j] = t;
        }
    }
}
```

The module statement defines the module name. The functions inside this module are described by the "Define" statement The definitions of the arguments are similar to those in C language, but some differences exist, as follows.

- You can specify whether a variable is input (*mode_in* or *IN*) or output (*mode_out* or *OUT*).
- For an array, you can specify its size with an input parameter.
- It is impossible to use the return value of the function itself.

After the Define statement, you can describe any program in C language inside of brace ({...}). Argument names can be used as the arguments in C language without modification.

You can call a library function, like the example, below.

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

This above example calls "`mmul`" function, which writes in C language.

And, you can define multiple functions inside the module's definition file. These descriptions should be in the remote executable program.

At the time of this writing, fundamental data types and their arrays are supported. We are going to support structure data types in a future release.

Details of IDL

A description of IDL includes some elements.

- Module statement
- Define statement
- Globals statement
- FORTRAN format statement

In the explanation below, we define the identifier as names which consist of alphanumeric characters with start with the element followed by an underscore ('_').

Module statement

We define the module's name.

```
Module module_name;
```

`module_name` is the identifier of the module. You should define the module name first in IDL file.

Define statement

We define an interface of a function which called from the remote program.

```
Define function_name (parameter1,parameter2,...)
"...description..."
interface_body
```

Function name is "`function_name`." We describe this parameter as follows.

```
 mode  type_specifier  parameter_name
```

Mode specifies whether an argument is input or output. If the argument is input, you write "*mode_in*" or "*IN*." If the argument is output, you should write "*mode_out*" or "*OUT*." And, if you want to allocate temporary data, you can specify "*work*." the "*type_specifier*" supports the names of the fundamental data type in C language, "*string*," which stands for string, "*filename* which stands for file specifyed filename and "*filepointer*" which stands for filepointer.

You can specify arguments of an array in C language.

```
 mode  type_specifier  parameter_name[size]...
```

Arguments of a multi-dimensional array are enclosed in brackets ([...]) for each dimension, as in C language.

You can describe the upper limit, bottom limit and stride of a transferred array area.

```
 mode  type_specifier  parameter_name[size:low,high,stride]...
```

Between the body information about the function is described in a string.

In the body, you need to conform to C language.

```
Define function_name (...) { in manner of C language }
```

In the description of a function of C , arguments are accessed as a parameter variable.

And, if you call a function which is linked, you write the function call after the Call directive.

```
Define function_name (...) Calls foo(...);
```

Globals statement

You can describe in programs written in C language the functions and data which are to be used in an entire module. For example, you can describe a necessary function definition when you define the definition of the function in C language. And, you can describe variables which are shared in functions and multiple functions.

```
Globals { ... any programs }
```

Fortranformat statement

This specifies the rule of function mangling when function link to FORTRAN program.

Differences from Ninf IDL

OmniRPC's IDL description is based on Ninf's IDL. However, there are some differences.

- Ninf creates a remote executable program for each definition of a function. OmniRPC creates a remote executable program for the entire module.
- In OmniRPC, variables which are defined in a Global statement are shared in a function inside the module.
- Ninf dose not have "Required" specifications which specify the link to a library for a function.
- As of now, OmniRPC's IDL cannot define remote functions for MPI.

IDL grammar

We show the informal definitions of IDL grammar as follows.

- '...' indicates a literal.
- IDENTIFIER is an identifier, CONSTANT is a constant.
- STRING is one or more character inside double quotation marks("..."). OPT_STRING is a STRING which can be omitted.
- {..}* stands iteration which not less than 0.
- C_PROGRAM stands for any program in C.
- type_specifier are fundamental data types of C, string, filename which stands for file and filepointer which stands for FILE pointer.

```
program := {declaration}*

declaration:=
          'Module' IDENTIFIER ';'
| 'Define' interface_definition OPT_STRING interface_body
        | 'Globals' '{' C_PROGRAM '}'
        | 'Fortranformat' STRING ';'
;

interface_definition:=
     IDENTIFIER '(' parameter {',' parameter}* ')'
;

parameter:= decl_specifier declarator ;

decl_specifier:
type_specifier
| MODE
| MODE type_specifier
| type_specifier MODE
| type_specifier MODE type_specifier
```

```
;

MODE := 'mode_in' | 'IN' | 'mode_out' | 'OUT';

declarator=:
  IDENTIFIER
| '(' declarator ')'
| declarator '['expr_or_null ']'
| declarator '['expr_or_null ':' range_spec ']'
| '*' declarator
;

range_spec=:
     expr
| expr ',' expr
| expr ',' expr ',' expr
;

interface_body:
'{' C_PROGRAM '}'
| CALLS OPT_STRING IDENTIFIER '(' IDENTIFIER {',' IDEFINTIER}* ')' ';'
;

expr_or_null:=  expr | /* null */;

expr:=   primary_expr
| '*' expr /* pointer reference */
| '-' expr /* unary minus */
| expr '/' expr
| expr '%' expr
| expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '^' expr
| expr RELOP expr
        | expr '?' expr ':' expr
;

primary_expr:=
 primary_expr '[' expr ']'
| IDENTIFIER
| CONSTANT
| '('  expr  ')'
;
```

Appendix C. Command Reference

Command Reference

omrpc-register

Name

`omrpc-register` — Operations about OmniRPC's registry.

Synopsis

omrpc-register [-path *reg_path*] [-show | -help | -clear | [-register | -remove] *rex_prog*]

Description

Adds registry of remote executable programs, and removes them from the registry. Shows the current registry.

Options

-path reg_path

Specifies path to the registry. The default registry path is "$HOME/.omrpc-registry".

-show

Shows information about the current registry.

-clear

Removes all information about current registry.

-register *rex_prog*

Registers the remote executable program *rex_prog* on remote host.

-remove *rex_prog*

Removes the remote executable program *rex_prog* from the registry.

-help

Displays help and exits.

omrpc-cc

Name

`omrpc-cc` — Compile driver for OmniRPC program.

Synopsis

omrpc-cc [options] [*`file.idl`*] [*`file.c ...`*]

Description

The compile driver for the OmniRPC program. Links to necessary OmniRPC library. And, if the IDL file is in the arguments, creates remote executable program which corresponds to it.

When you create client program, the command is used like a C compiler. For example:

```
% omrpc-cc -o foo foo.c goo.o
```

Like a C compiler, this compiles `foo.c`, and links to `goo.o`, then the executable file foo is created. During this time, this command automatically sets the library and include file, which are necessary for OmniRPC. Additionally, if "`-o`" is not specified, like a C compiler, the object file is "`a.out`."

If the IDL file has an ".idl" extension in the argument file, this command generates the OmniRPC remote executable program from this file. This program name takes on the extension ".rex", for instance,

```
% omrpc-cc bar.idl goo.o
```

In this case, bar.rex is generated from bar.idl, and goo.o is the library linked to a remote execution program.

Options

-c

Compiles and generates the object file, but not the link.

-o

Specifies object files.

-l`lib_dir`

Specifies the path to the library's directory.

-show

Shows the command which should be executed. Execution is not carried out.

-echo

Shows the command which should be executed.

-liner `link_command`

Specifies the command to the link. If this option is omitted, the default command is **cc**, which is a driver of the C compiler. For example, if you use Fortran, you can specify **f77** as a linker.

omrpc-fc

Name

`omrpc-fc` — FORTRAN Compile driver for OmniRPC program.

Synopsis

omrpc-fc [options] [*`file.idl`*] [*`file.f ...`*]

Description

The compile driver for the OmniRPC program. Links to necessary OmniRPC library. And, if the IDL file is in the arguments, creates remote executable program which corresponds to it.

When you create client program, the command is used like a FORTRAN compiler. For example:

```
% omrpc-fc -o foo foo.f goo.o
```

Like a FORTRAN compiler, this compiles `foo.f`, and links to `goo.o`, then the executable file foo is created. During this time, this command automatically sets the library and include file, which are necessary for OmniRPC. Additionally, if "`-o`" is not specified, like a FORTRAN compiler, the object file is "`a.out`."

If the IDL file has an ".idl" extension in the argument file, this command generates the OmniRPC remote executable program from this file. This program name takes on the extension ".rex", for instance,

```
% omrpc-fc bar.idl goo.o
```

In this case, bar.rex is generated from bar.idl, and goo.o is the library linked to a remote execution program.

Options

-c

Compiles and generates the object file, but not the link.

-o

Specifies object files.

-l`lib_dir`

Specifies the path to the library's directory.

-show

Shows the command which should be executed. Execution is not carried out.

-echo

Shows the command which should be executed.

-liner `link_command`

Specifies the command to the link. If this option is omitted, the default command is **cc**, which is a driver of the C compiler. For example, if you use Fortran, you can specify **f77** as a linker.

-fc `fortran_compiler`

If you want to use another compiler, use the "-fc" option (default FORTRAN compiler is g77).

omrpc-register

Name

`omrpc-gen` — A program to generate OmniRPC's stub.

Synopsis

omrpc-gen [*file.idl*] [*file.c*]

Description

Generates the stub program `file.c` (in C language) from the IDL file `file.idl`. If the IDL file is omitted, the input is from *STDIN*. If the stub program argument is omitted, the output is *STDOUT*. Usually IDL files uses the file extension ".idl". The source file of stub program use the extension ".rex.c". The remote executable program which is compiled from the stub program, use the extension ".rex".

This program usually call from the OmniRPC's compile driver, so you can use it directly. If you compile the stub source program directly, you should specify the include directory, which is below the install directory, as the include path. And when you create the remote executable program, you should link the library for the stub (libomrpc_stub.a and libomrpc_io.a). If you install OmniRPC at the INSTALL_DIR, you should compile in the way shown below.

```
%cc -o foo.rex -IINSTALL_DIR/include foo.rex.c
    ... -LINSTALL_DIR/lib -lomrpc_stub.a -lomrpc_io
```

Appendix D. OmniRPC API index

OmniRPC C API

OmniRpcInit

Name

`OmniRpcInit` — Initialization of OmniRPC system

Synopsis

```
#include <OmniRpc.h>
void OmniRpcInit(int *argc, char **argv[]);
```

Description

Initializes OmniRPC System. Sets *argc* by the *argc* pointer of the main function, sets argv by the argv pointer of it. This should be called before processing the argument information. In initialization, it uses the hostfile which is specified in the command line "--hostfile" or the hostfile on "$HOME/.omrpc-registry/hosts.xml" if not specified. It reads the hostfile and executes omrpc-agent on the host which are used. And, it reads registry information about the remote executable programs which are registered on the each host.

OmniRpcFinalize

Name

`OmniRpcInit` — Finalize OmniRPC system

Synopsis

```
#include <OmniRpc.h>
void OmniRpcFinalize(void);
```

Description

Finalize OmniRPC system. Terminate all remote executable programs and agents.

OmniRpcCall

Name

`OmniRpcCall` — Synchronous call of remote function

Synopsis

```
#include <OmniRpc.h>
int OmniRpcCall(const char *entry_name, ...);
```

Description

Calls a remote function which is specified by *`entry_name`*. It blocks the caller thread until the end of the function call.

It searches for the function name from modules which are registered on the one of remote hosts which are described in the hostfile, and calls the function on the appropriate remote host. If the function name is not found, it returns OMRPC_ERROR. If it ends normally, it returns OMRPC_OK.

OmniRpcCallAsync

Name

`OmniRpcCallAsync` — Asynchronous call of remote function

Synopsis

```
#include <OmniRpc.h>
OmniRpcRequest OmniRpcRequest(const char *entry_name, ...);
```

Description

Requests to call a remote function which is specified by `entry_name`, and returns the data structure (OmniRpcRequest) for the request. It blocks the caller thread.

It search for a function name from the modules which are registered on the remote hosts, which are described in the hostfile, and calls the function on the appropriate remote host. If the function name is not found, it returns NULL.

OmniRpcWait

Name

`OmniRpcWait` — Wait for asynchronous call

Synopsis

```
#include <OmniRpc.h>
void OmniRpcWait(OmniRpcRequestreq);
```

Description

It blocks the caller thread until the end of an asynchronous call for the call request of *`req`*.

OmniRpcProbe

Name

`OmniRpcProbe` — Probing asynchronous function call

Synopsis

```
#include <OmniRpc.h>
int OmniRpcProbe(OmniRpcRequestreq);
```

Description

Probes whether or not an asynchronous function call which associated to the call request of *`req`* finishes. If it does not finish, it returns 0. If it finishes, it returns 1.

OmniRpcWaitAll

Name

`OmniRpcWaitAll` — Wait of multiple asynchronous calls

Synopsis

```
#include <OmniRpc.h>
voidOmniRpcWaitAll(intn, OmniRpcRequestreqs[]);
```

Description

It blocks caller threads until the end of all asynchronous function calls which correspond to *n* call requests of *reqs*. The call requests are stored in an array of OmniRpcRequest.

OmniRpcWaitAny

Name

`OmniRpcWaitAny` — Wait of multiple asynchronous function calls

Synopsis

```
#include <OmniRpc.h>
intOmniRpcWaitAny(intn, OmniRpcRequestreqs[]);
```

Description

It blocks caller threads until the end of one asynchronous function call which corresponds to *n* call requests of *req*, which are stored in an array of OmniRpcRequest. It returns the positions of finished call requests in the array. The finished call request elements on the array are set with NULL.

OmniRpcCreateHandle

Name

`OmniRpcCreateHandle` — Invocation of remote executable program

Synopsis

```
#include <OmniRpc.h>
OmniRpcHandle OmniRpcCreateHandle(const char *hostname, const char
*module_name);
```

Description

Executes the remote executable program of the module which is specified by *module_name*, and that is on the remote host which is specified by *hostname*. It returns handle corresponding to it. By using this, it calls functions on running remote executable programs with OmniRpcCallByHandle. If *hostname* is NULL, it selects the appropriate host in registered modules, and execute remote executable program of module. If host name or module are incorrect, it returns NULL.

OmniRpcCallByHandle

Name

`OmniRpcCallByHandle` — Synchronous function call with OmniRpcHandle

Synopsis

```
#include <OmniRpc.h>
int OmniRpcCallByHandle(OmniRpcHandlehandle, char *entry_name, ...);
```

Description

Calls function on running remote executable program corresponding to the *handle* by OmniRpcCreateHandle. It blocks the caller thread until the end of the function call. If the function does not exist, it returns NULL.

OmniRpcCallAsyncByHandle

Name

`OmniRpcCallAsyncByHandle` — Asynchronous call of function with OmniRpcHandle

Synopsis

```
#include <OmniRpc.h>
OmniRpcRequest OmniRpcCallAsyncByHandle(OmniRpcHandlehandle, char
*entry_name, ...);
```

Description

Calls function on running executable program corresponding to the *`handle`* created by OmniRpcCreateHandle. It calls a function and returns the OmniRpcRequest which is associated with it. It probes the end of the function and blocks control by OmniRpcWait, OmniRpcProbe, OmniRpcWaitAll and OmniRpcWaitAny APIs.

OmniRpcDestroyHandle

Name

`OmniRpcDestroyHandle` — Termination of remote executable program by OmniRpcHandle

Synopsis

```
#include <OmniRpc.h>
void OmniRpcDestroyHandle(OmniRpcHandlehandle);
```

Description

Terminates running remote executable program corresponding to the *`handle`* which is created by OmniRpcCreateHandle.

OmniRpcModuleInit

Name

`OmniRpcModuleInit` — Setting of module initialization

Synopsis

```
#include <OmniRpc.h>
intOmniRpcModuleInit(const char *module_name, ...);
```

Description

Sets arguments for initialization of the module named *`module_name`*. The "Initialize" function is required in the modules of remote executable programs. When the remote executable programs of modules are executed, it calls the "Initialize" function with sets the arguments. This API only sets the arguments; the actual initialization occurs when the remote executable program is executed.

OmniRpcExecInit

Name

`OmniRpcExecInit` — Initialization for direct invocation of remote executable program

Synopsis

```
#include <OmniRpc.h>
voidOmniRpcExecInit(int *argc, char **argv[]);
```

Description

Initializes OmniRPC system for direct invocation of a remote executable program. You should set *`argc`* by the *`argc`* pointer of the main function, and sets argv by the argv pointer of it. You call this API in the main function of program before processing argument information. As in OmniRpcInit, the agent is not executed.

In the command option, if you specify "--globus", it uses GRAM, or, if you specify "--ssh", it uses ssh to invoke the remote executable program directly. It uses rsh by default.

OmniRpcExecFinalize

Name

`OmniRpcExecFinalize` — Termination of OmniRPC system for direct invocation of remote executable program

Synopsis

```
#include <OmniRpc.h>
void OmniRpcExecFinalize(void);
```

Description

Finalizes of OmniRPC system. Terminates all remote executable programs.

OmniRpcExecOnHost

Name

`OmniRpcExecOnHost` — Direct invocation of remote executable program

Synopsis

```
#include <OmniRpc.h>
OmniRpcExecHandleOmniRpcExecOnHost(char *host_name, char *prog_name);
```

Description

Invokes remote executable program *`prog_name`* on the remote host which is specified *`host_name`*, and returns the corresponding handle. Specify prog_name with path on remote host. Using this with OmniRpcExecCall, it is possible to call functions on running remote executable programs. You should initialize with OmniRpcExecInit API if you use this function. If the host name or module name is incorrect, it returns NULL.

OmniRpcExecCall

Name

`OmniRpcExecCall` — Synchronous call by OmniRpcExecHandle

Synopsis

```
#include <OmniRpc.h>
int OmniRpcExecCall(OmniRpcHandlehandle, const char *func_name, ...);
```

Description

Calls a function of the running remote executable program corresponding to the *`handle`* created by OmniRpcExecOnHost. It blocks the called thread until the end of the function. If the remote executable program does not have a function, it returns NULL.

OmniRpcExecTerminate

Name

`OmniRpcExecTerminate` — Termination of remote executable program by OmniRpcExecHandle

Synopsis

```
#include <OmniRpc.h>
voidOmniRpcExecTerminate(OmniRpcHandlehandle);
```

Description

Terminates the running remote executable program corresponding to the *`handle`* created by OmniRpcExecOnHost.

OmniRPC FORTRAN API

All APIs are wrapper subroutines of OmniRPC.

OMNIRPC_INIT

Name

`OMNIRPC_INIT` — Initialization of OmniRPC system

Synopsis

```
callOMNIRPC_INIT(void);
```

Description

Initializes the OmniRPC system.

OMNIRPC_FINALIZE

Name

`OMNIRPC_FINALIZE` — Finalizing OmniRPC system

Synopsis

```
call OMNIRPC_FINALIZE(void);
```

Description

Finalizes the OmniRPC system. Terminates all remote executable programs and agents.

OMNIRPC_CALL

Name

`OMNIRPC_CALL` — Synchronous call of remote function

Synopsis

```
character*(*) entry_name



call OMNIRPC_CALL(entry_name);
```

Description

Calls a remote function which is specified by entry_name. It blocks the caller thread until the end of the function call. entry_name is a string to which "*" is added.

It searches for the function name from modules which are registered on the remote hosts described in the hostfile, and calls the function on the appropriate remote host.

OMNIRPC_CALL_ASYNC

Name

`OMNIRPC_CALL_ASYNC` — Asynchronous call of remote function

Synopsis

```
character*(*) entry_name
integer ireq
```

```
call OMNIRPC_CALL_ASYNC(ireq, entry_name, ...);
```

Description

Makes a request to call the remote function which is specified entry_name, and returns the ID of req for the request. entry_name is a string which is a combination of the function name and '*' character. It blocks the caller thread. It probes the end of the function and blocks control by using OMNIRPC_WAIT, OMNIRPC_PROBE, OMNIRPC_WAIT_ALL, OMNIRPC_WAIT_ANY API, all of which are possible whether the function ends or not.

Searches for the function name from modules registered on remote hosts, which are described in the hostfile, and calls function on the appropriate remote host. If the function name is not be found, it sets req to 0.

OMNIRPC_WAIT

Name

`OMNIRPC_WAIT` — Wait for asynchronous call

Synopsis

```
integer ireq
```

```
call OMNIRPC_WAIT(ireq);
```

Description

It blocks the caller thread until the end of an asynchronous call for the call request of ireq.

OMNIRPC_PROBE

Name

`OMNIRPC_PROBE` — Probing asynchronous function call

Synopsis

```
integer ireq, istatus
```

```
call OMNIRPC_PROBE(ireq, istatus);
```

Description

Probes whether or not an asynchronous function call which is associated with call the request of ireq finishes. If it does not finish, it sets istatus as 1. If it finishes, it sets istatus as 0.

OMNIRPC_WAIT_ALL

Name

`OMNIRPC_WAIT_ALL` — Wait of multiple asynchronous calls

Synopsis

```
integer nreq, ireqs(*)
```

```
call  OMNIRPC_WAIT_ALL(nreq, ireqs);
```

Description

It blocks caller threads until the end of all asynchronous function calls corresponding to nreq. Calls requests of req which are stored in an array of ireqs.

OMNIRPC_WAIT_ANY

Name

`OMNIRPC_WAIT_ANY` — Wait of multiple asynchronous function calls

Synopsis

```
integer nreq, ireqs(*), iret
```

```
call OMNIRPC_WAIT_ANY(nreq, ireqs, iret);
```

Description

It blocks caller threads until the end of one asynchronous function call corresponding to nreq call requests of ireq, which are stored in an array of OmniRpcRequest. As a return value, it sets iret to the position of the finished call requests in the array. Finished call request elements in the array is set as 0.

OMNIRPC_CREATE_HANDLE

Name

`OMNIRPC_CREATE_HANDLE` — Execution of remote executable program

Synopsis

```
integer ihandle
character*(*) host_name, module_name



call OMNIRPC_CREATE_HANDLE(ihandle, host_name, module_name);
```

Description

Executes the remote executable program of the module specified as module_name, on the remote host specified as host_name. It returns the corresponding ihandle. It calls functions on running remote executable programs with OmniRpcCallByHandle. The host_name and module_name have "*" attached to the hostname which is used or to the module. If the host_name has a "*", it selects the appropriate host fro the registered modules, and executes the remote executable program of the module. If the host name or module is incorrect, it sets ihandle as 0.

OMNIRPC_CALL_BY_HANDLE

Name

`OMNIRPC_CALL_BY_HANDLE` — Synchronous function call with OmniRpcHandle

Synopsis

```
integer ihandle
character*(*) entry_name
```

```
call  OMNIRPC_CALL_BY_HANDLE(ihandle, entry_name, ...);
```

Description

Calls entry_name function on the running remote executable program corresponding to the ihandle created by OmniRpcCreateHandle. A "*" is added the the entry_name function name. It blocks the caller thread until the end of the function call.

OMNIRPC_CALL_ASYNC_BY_HANDLE

Name

OMNIRPC_CALL_ASYNC_BY_HANDLE — Asynchronous call of function with OmniRpcHandle

Synopsis

```
integer ireq,ihandle
character*(*) entry_name
```

```
call OMNIRPC_CALL_ASYNC_BY_HANDLE(ireq, ihandle, entry_name, ...);
```

Description

Calls the entry_name function on the running remote executable program corresponding to the ihandle created by OmniRpcCreateHandle. entry_name is a string which combines the function name and the "*" character. Probes the end of the function and blocks control by using OMNIRPC_WAIT, OMNIRPC_PROBE, OMNIRPC_WAIT_ALL, OMNIRPC_WAIT_ANY API, all of which are possible whether the function end or not.

OMNIRPC_DESTROY_HANDLE

Name

`OMNIRPC_DESTROY_HANDLE` — Termination of remote executable program by OmniRpcHandle

Synopsis

```
integer ihandle



call OMNIRPC_DESTROY_HANDLE(ihandle);
```

Description

Terminates running remote executable program corresponding to the ihandle which is created by OmniRpcCreateHandle.

OMNIRPC_MODULE_INIT

Name

`OMNIRPC_MODULE_INIT` — Setting for module initialization

Synopsis

```
character*(*) module_name



call OMNIRPC_MODULE_INIT(module_name, ...);
```

Description

Sets arguments for initialization of the module module_name. A "*" is added to module_name. The "Initialize" function must be required in the module of the remote executable program. When the remote executable programs of the modules are executed, it calls the "Initialize" function with set arguments. This API is only set; actual initialization occurs when the remote executable program is executed.

Appendix E. FAQ

Q: Hostname of the client host cannot be accessed.

A: In the OmniRPC system, after the agent's invocation, the program on the remote host requests access to the client host. Therefore, it is necessary for the remote host to know the hostname which can be accessed to the client host By default, the OmniRPC system uses the hostname through hostname commands. But, in some settings, there may exits a hostname which cannot be accessed from an outside network exists. You should set FQDN(Full Qualified Domain Name) by the hostname command or environment variable OMRPC_HOSTNAME as FDQN.

```
(csh or tcsh)
% setenv OMRP_HOSTNAME  FQDN

(bash)
$ export OMRP_HOSTNAME  FQDN
```

There is the same problem in the Globus Toolkit environment. In this case, set the environmental variable GLOBUS_HOSTNAME to FQDN. for more details, please see Globus information.

Q: Our cluster does not support **RSH** due to security reason.

A: Usually cluster nodes accept **RSH**, So, the agent can invoke the remote executable program on the cluster nodes. But in some situations, the cluster nodes restrict the use of **RSH** and support **SSH**. Therefore, you can use **SSH** to invoke remote executable programs. You should write explicitly to use **SSH**. You can change the cluster nodes file (which is introduced in Use of a built-in round-robiin scheduler) as below.

```
hpc1 ssh
hpc2 ssh
hpc3 ssh
```

If **ssh** is omitted in the above description, **rsh** is used.

Q: I use SSH both for agent invocation and for worker invocation when I want to use clusters. But An error has occured when invocation of workers.

A: It seems that authentications between OmniRPC agent and worker program are failed. Because OmniRPC agent cannot use the ssh-agent's pass-phrase in which client program runs. As result, when OmniRPC agent uses ssh to invoke worker program in cluster nodes, authentication between agent and worker program is failed. Easy way to solve this issue is to add ssh option in "`.ssh/config`" as follows.

```
ForwardAgent yes
```

Q: After I lunched the client program, client program exits with like bellow message.

```
OMRPC_FATAL(localhost:./[programname]): omrpc_io_accept: time out
```

A: Authentication phase sometimes takes more than 15 seconds when client program invokes OmniRPC agent in remote nodes. OmniRPC's default timeout is 15 seconds. If you want increase this number, please set the TimeOut element in hostfile like bellow example.

```
<?xml version="1.0" ?>
<OmniRpcConfig>
   <Host name="jones.is.tsukuba.ac.jp">
   <TimeOut second="20">
</OmniRpcConfig>
```