Users' Guide to GridSolve
Version 0.15

Jack Dongarra, Keith Seymour, Asim YarKhan

Innovative Computing Laboratory
Department of Computer Science
University of Tennessee

May 2006

Users' Guide to GridSolve: (http://icl.cs.utk.edu/gridsolve/)
by Sudesh Agrawal, Jack Dongarra, Kiran Sagi, Keith Seymour, Asim YarKhan

Contents

List of Figures

Chapter 1

Overview of GridSolve

1.1 An Introduction to Distributed Computing

The efficient solution of large problems is an ongoing thread of research in scientific computing. An increasingly popular method of solving these types of problems is to harness disparate computational resources and use their aggregate power as if it were contained in a single machine. This mode of using computers that may be distributed in geography, as well as ownership, has been termed Distributed Computing. Some of the major issues concerned with Distributed Computing are resource discovery, resource allocation and resource management, fault-tolerance, security and access control, scalability, flexibility and performance. Various organizations have developed mechanisms that attempt to address these issues, each with their own perspectives of how to resolve them.

1.2 What is GridSolve?

GridSolve (http://icl.cs.utk.edu/gridsolve) is an example of a Distributed Computing system that hopes to present functionalities and features that a wide variety of scientists will find highly useful and helpful.

1.2.1 Background

Various mechanisms have been developed to perform computations across diverse platforms. The most common mechanism involves software libraries. Unfortunately, the use of such libraries presents several difficulties. Some software libraries are highly optimized for only certain platforms and do not provide a convenient interface to other computer systems. Other libraries demand considerable programming effort from the user. While several tools have been developed to alleviate these difficulties, such tools themselves are usually available on only a limited number of computer systems and are rarely freely distributed. Matlab [TMW92] and Mathematica [Wol96] are examples of such tools. These considerations motivated the establishment of the GridSolve project. The basic philosophy of GridSolve is to provide a uniform, portable and efficient way to access computational resources over a network.

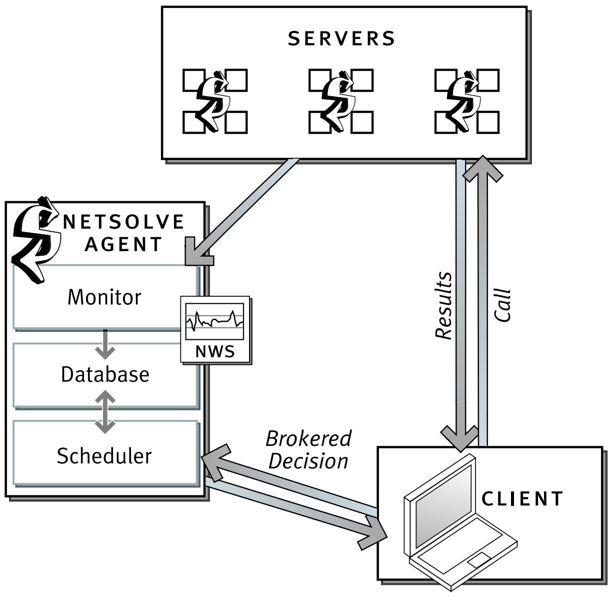

Figure 1.1: Overview of GridSolve

1.2.2 Overview and Architecture

The GridSolve project is being developed at the University of Tennessee's Computer Science Department. It provides remote access to computational resources, both hardware and software. Built upon standard Internet protocols, like TCP/IP sockets, it is available for all popular variants of the UNIX™ operating system, and parts of the system are available for the Microsoft Windows 2000™ and Windows XP™ platforms. The GridSolve system is comprised of a set of loosely connected machines. By loosely connected, we mean that these machines are on the same local, wide or global area network, and may be administrated by different institutions and organizations. Moreover, the GridSolve system is able to support these interactions in a heterogeneous environment, i.e. machines of different architectures, operating systems and internal data representations can participate in the system at the same time.

Figure 1.1 shows the global conceptual picture of the GridSolve system. In this figure, we can see the three major components of the system: the *client*, the *agent*, and the *servers* (computational or software resources). GridSolve and systems like it are often referred to as Grid Middleware. GridSolve acts as a glue layer that brings the application or user together with the hardware and/or software it needs to complete useful tasks. At the top tier, the GridSolve client library is linked in with the user's application. The application then makes calls to GridSolve's application programming interface (API) for specific services. Through the API, GridSolve client-users gain access to aggregate resources without needing to know anything about computer networking or distributed computing. In fact, the user does not even have to know remote resources are involved. The GridSolve agent maintains a database of GridSolve servers along with their capabilities (hardware performance and allocated software) and dynamic usage statistics. It uses this information to allocate server resources for client requests. The agent finds servers that will service requests the quickest, balances the load amongst its servers and keeps track of failed ones. The GridSolve server is a daemon process that awaits client requests. The server can run on single workstations, clusters of workstations, symmetric multi-processors or machines with massively parallel processors. A key component of the GridSolve server is a source code generator which parses a GridSolve Interface Definition Language (gsIDL) file. This gsIDL contains information that allows the GridSolve system to create new modules and incorporate new functionalities. In essence, the gsIDL defines a wrapper that GridSolve uses to call functions being incorporated. The (hidden) semantics of a GridSolve request are:

1. Client contacts the agent for a list of capable servers.
2. Client contacts server and sends input parameters.
3. Server runs appropriate service.
4. Server returns output parameters or error status to client.

From the user's perspective, the call to GridSolve acts just like the call to the original function.

1.2.3 Who is the GridSolve User?

There are two types of GridSolve users. The first type of user is one who installs and accesses only the client interface(s) and utilizes existing pools of resources (agent(s) and server(s)). The second type of GridSolve user installs and administrates his own GridSolve system (client, agent(s), server(s)), and potentially enables his software to be used by GridSolve. This Users' Guide addresses the needs of both types of users. Note that the term "administrates" or "administrator" here simply refers to the person setting up and maintaining the GridSolve agent and server components – no superuser privileges are needed to install or use any component of the GridSolve system.

Chapter 2

Downloading, Installing, and Testing

The GridSolve client software is available for UNIX and UNIX-like operating systems and Windows environments. All of the client, agent, and server software is bundled into one tar-gzipped file for UNIX-like operating systems. There is a separate distribution file for the Windows client. No root/superuser privileges are needed to install or use any component of the GridSolve system. GridSolve uses autoconf to create a build environment that is similar to most other Open Source projects.

2.1 Installation on Unix Systems

The GridSolve distribution tar file is available from the GridSolve web site located at the following URL:

```
http://icl.cs.utk.edu/gridsolve/software/index.html
```

After uncompressing the source code, go to the root of the GridSolve source tree and using the provided configure script, simply do:

```
% ./configure
% make
% make check
```

There are a few GridSolve-specific options that may be specified when running configure:

- `--with-blas`: this specifies the location of the BLAS library. If the library is in a standard location, this does not need to be specified, but `--with-blas=/nonstandard/lib/libblas.a` would be used to specify a nonstandard location.
- `--with-lapack`: this specifies the location of the LAPACK library. If the library is in a standard location, this does not need to be specified, but you can specify a nonstandard location with `--with-lapack=/nonstandard/lib/liblapack.a`.
- `--with-matlab`: specifies the location of the Matlab installation. You may specifiy `--with-matlab=no` to disable the Matlab client.
- `--with-dsi-ibp`: enables DSI and specifies the location of the IBP library to use for DSI. See Chapter 12 for more information about using the DSI API.

- `--enable-debug`: if enabled, this causes debugging output to be printed to the console.
- `--enable-profiling`: enables client profiling of the various stages in the procedure call (e.g. contacting agent, sending data, etc.). See Chapter 13 for more information about the profiling interface.

Use `autoreconf` if you need to regenerate the Autotools files. This should only be necessary if you obtain the code from the CVS repository. You will need a relatively new version of the Autotools tools.

```
% autoreconf
```

When building for multiple architectures:

```
% mkdir ‘config_ac/config.guess‘
% cd ‘config_ac/config.guess‘
% ../configure
% make
% make check
```

For Solaris, this might be an example for a make from the CVS for the builders. This disables dependency tracking because Solaris make/cc may have trouble with it. When building from a release version, you should not need the disable-dependency-tracking flag because the dependency information is hardwired into the Makefiles.

```
% mkdir sparc-sun-solaris2.8
% cd sparc-sun-solaris2.8
% ../configure -C --enable-debug --disable-dependency-tracking
% make -k
% make -k check
```

Note that the ”make check” part of the build procedure is not intended to test anything. It is used to build the GridSolve services. If you are only going to use the client, it is not necessary.

2.2 Testing the Unix Installation

Testing solely the client software means that a pre-existing GridSolve system will be contacted, possibly the default agent and servers running at the University of Tennessee. That system can be contacted via the host `gridsolve.cs.utk.edu` which should always be running an agent. The step-by-step procedure to test your GridSolve client installation is as follows:

```
% cd GridSolve/src/testing/gridrpc
% setenv GRIDSOLVE_AGENT gridsolve.cs.utk.edu
% ./totaltest
```

While the tester is running, it prints messages about its execution. This test tests only the GridRPC interface. Similar tests for the NetSolve compatibility interfaces (C and Fortran77) exist in the `src/testing/netsolve` directory. Details of this process are explained in the following chapters. For more information on the C and Fortran77 interfaces, see Chapter 4. Chapter 5 describes how to test the Matlab interface.

2.3 Installation on Windows Systems

This section describes the installation and testing of the Windows version of the GridSolve client software. At present, the software is distributed in the form of a self-extracting exe file. The Windows client only works with Windows 2000™ and Windows XP™. It will not run on Windows 98™ or earlier. The contents of the self-extracting exe file are as follows, where `GRIDSOLVE_DIR` refers to the directory where you have unzipped the distribution.

- `GRIDSOLVE_DIR\` – This directory contains the readme file and an installation script.
- `GRIDSOLVE_DIR\lib` – This directory contains the GridSolve client library.
- `GRIDSOLVE_DIR\matlab` – This directory contains the Matlab binaries.
- `GRIDSOLVE_DIR\tools` – This directory contains various tools for managing GridSolve.
- `GRIDSOLVE_DIR\testing` – This directory contains various sample binary test programs that you can run to verify your installation.

The installation process is quite simple.

1. Run the exe you downloaded from the GridSolve webpage to extract the files to a directory.
2. Then run the executable `gridsolve_install.exe` to set the registry keys for GridSolve.

To determine the agent host name, the user can issue the following commands from a DOS prompt:

```
C:\> cd GRIDSOLVE_DIR\tools
C:\> getagent
```

To set a new agent host name, the user must issue the following command:

```
C:\> cd GRIDSOLVE_DIR\tools
C:\> setagent [agent host name]
```

If the agent host name is not specified on the command line, you will be prompted for a host name. You will have the option of specifying a name or accepting the current agent name set in the registry.

The de-installation process is quite similar.

```
C:\> cd GRIDSOLVE_DIR
C:\> gridsolve_install -uninstall
  [The above program removes the keys from the Windows registry]
C:\> rmdir /s GRIDSOLVE_DIR
```

2.4 Testing the Windows installation

You can use the various programs in the `GRIDSOLVE_DIR\testing` directory to test your GridSolve installation. Remember that a valid GridSolve agent and server should already be running, and the required problems should be installed on the servers. Here is a list of the test programs currently available:

- c_totaltest
- c_totaltest_async
- f_totaltest
- f_totaltest_async

For example, to perform a sample run of c_totaltest, the user must do the following:

1. Use `setagent` to point to the correct agent host. (e.g. `setagent gridsolve.cs.utk.edu`)
2. Run `c_totaltest.exe` from the testing directory.

2.5 Using GridSolve from Windows Matlab

A user new to Gridsolve will find the Matlab interface very simple. The matlab interface is in `GRIDSOLVE_DIR\matlab`. To access the interface:

1. Start up Matlab
2. Click on File ▷ Set Path ...
3. Add the `GRIDSOLVE_DIR\matlab` directory to the path

The interface consists of 4 GridSolve dlls, which should be in your Matlab path: `gridsolve.dll`, `gridsolve_nb.dll`, `gridsolve_err.dll`, and `gridsolve_errmsg.dll`.

To begin testing from Matlab, there are a couple of commands that display the status of the system. The following command prints the agent and servers currently available:

```
gridsolve('?')
```

Specifying the same function name without any arguments will print the list of problems that can be solved:

```
gridsolve
```

Chapter 3

GridRPC API

3.1 Introduction

The primary API used by GridSolve is GridRPC, a standardized, portable, and simple programming interface for remote procedure call (RPC) over the Grid. In this section, we informally describe the GridRPC model and the functions that comprise the API. Appendix B contains a detailed listing of the function prototypes. Chapter 4 describes the NetSolve compatibility layer, which provides an API that matches the API of NetSolve 2.0.

3.2 Function Handles and Session IDs

Two fundamental objects in the GridRPC model are *function handles* and *session IDs*. The function handle represents a mapping from a function name to an instance of that function on a particular server. The GridRPC API does not dictate the mechanics of resource discovery since different underlying GridRPC implementations may use vastly different protocols. Once a particular function-to-server mapping has been established by initializing a function handle, all RPC calls using that function handle will be executed on the server specified in that binding. A session ID is an identifier representing a particular non-blocking RPC call. The session ID is used throughout the API to allow users to obtain the status of a previously submitted non-blocking call, to wait for a call to complete, to cancel a call, or to check the error code of a call.

3.3 Initializing and Finalizing Functions

The initialize and finalize functions are similar to the MPI initialize and finalize calls. Client GridRPC calls before initialization or after finalization will fail.

- `grpc_initialize` reads the configuration file and initializes the required modules.
- `grpc_finalize` releases any resources being used by GridRPC.

3.4 Remote Function Handle Management Functions

The *function handle management* group of functions allows creating and destroying function handles.

- `grpc_function_handle_default` creates a new function handle using the default server. This could be a pre-determined server name or it could be a server that is dynamically chosen by the resource discovery mechanisms of the underlying GridRPC implementation, such as the NetSolve agent.

- `grpc_function_handle_init` creates a new function handle with a server explicitly specified by the user.

- `grpc_function_handle_destruct` releases the memory associated with the specified function handle.

- `grpc_get_handle` returns the function handle corresponding to the given session ID (that is, corresponding to that particular non-blocking request).

3.5 GridRPC Call Functions

A GridRPC may be either blocking (synchronous) or non-blocking (asynchronous) and it accepts a variable number of arguments (like `printf`) depending on the calling sequence of the particular routine being called.

- `grpc_call` makes a blocking remote procedure call with a variable number of arguments.

- `grpc_call_async` makes a non-blocking remote procedure call with a variable number of arguments.

3.6 Asynchronous GridRPC Control Functions

The following functions apply only to previously submitted non-blocking requests.

- `grpc_probe` checks whether the asynchronous GridRPC call has completed.

- `grpc_probe_or` checks whether any of the previously issued non-blocking calls in a given set have completed.

- `grpc_cancel` cancels the specified asynchronous GridRPC call.

- `grpc_cancel_all` cancels *all* previously issued calls.

3.7 Asynchronous GridRPC Wait Functions

The following five functions apply only to previously submitted non-blocking requests. These calls allow an application to express desired non-deterministic completion semantics to the underlying system, rather than repeatedly polling on a set of sessions IDs. (From an implementation standpoint, such information could be conveyed to the OS scheduler to reduce cycles wasted on polling.)

- `grpc_wait` blocks until the specified non-blocking requests to complete.

- `grpc_wait_and` blocks until *all* of the specified non-blocking requests in a given set have completed.

- `grpc_wait_or` blocks until *any* of the specified non-blocking requests in a given set has completed.
- `grpc_wait_all` blocks until *all* previously issued non-blocking requests have completed.
- `grpc_wait_any` blocks until *any* previously issued non-blocking request has completed.

3.8 Error Reporting Functions

Of course it is possible that some GridRPC calls can fail, so we need to provide the ability to check the error code of previously submitted requests. The following error reporting functions provide error codes and human-readable error descriptions.

- `grpc_error_string` returns the error description string, given a numeric error code.
- `grpc_get_error` returns the error code associated with a given non-blocking request.
- `grpc_get_failed_sessionid` returns the session ID of the last invoked GridRPC call that caused a failure.

Chapter 4

NetSolve Compatibility Interface

4.1 Introduction

The C and Fortran77 client interfaces for NetSolve compatibility are compiled as part of the normal build process, so if you have followed the procedures outlined in Chapter 2, the following library should exist:

```
src/netsolve/libnetsolve.a
```

This library contains both the C and Fortran77 interfaces.

Before linking to one of these libraries, the user must include the appropriate header file in his program:

- `src/client/netsolve.h` in C programs
- `src/client/fnetsolve.h` in Fortran77 programs

The Fortran77 include file is not mandatory, but increases the source program readability by allowing calling subroutines to manipulate the NetSolve error codes by variable name rather than by integer value. See [AAB+02] for detailed information about using the NetSolve API. The compatibility layer included in GridSolve works the same as the original NetSolve API, but because GridSolve uses a different Interface Definition Language, the calling sequence should be structured according to the GridSolve mechanism. See Chapter 10 for more detail on determining the calling sequence.

Chapter 5

Matlab Interface

5.1 Introduction

GridSolve can be built with an optional Matlab client interface. This interface allows a Matlab user to transparently and easily use remote services from within the Matlab session. GridSolve handles all the details involved in sending the arguments to the appropriate server and fetching the results.

5.2 Building and Enabling the Matlab Interface

At this time, if a Matlab installation can be located during the configuration process, the Matlab GridSolve interface will be built by default. If you wish to build without Matlab, you can pass the option `--with-matlab=no` to the configure script.

In order to use GridSolve, certain files need to be on the Matlab search path. In a C style shell, the following will setup the correct path. You can also use the Matlab command `addpath` to setup the path.

```
setenv MATLABPATH ${MATLABPATH}:${GRIDSOLVE_ROOT}/${GRIDSOLVE_ARCH}/src/matlab_client
setenv MATLABPATH ${MATLABPATH}:${GRIDSOLVE_ROOT}/src/matlab_client
```

5.3 Matlab GridSolve API

The Matlab GridSolve interface closely matches the GridRPC API.

- `gs_info('service_name')`
  This call will return information about the service.
- `[output_args, ...] = gs_call('service_name, input_args, ...)`
  This will make a blocking call to a GridSolve server that can perform the service.
- `sessionid = gs_call_async('service_name, input_args, ...)`
  This will make a asynchronous non-blocking call to a GridSolve server that can perform the service. The sessionid is used to probe the call and to wait for results.
- `status = gs_call_probe(sessionid)`
  This is used to probe an asynchronous call to see if it has completed. It returns 1 if the call has completed.

- `[output_args, ..] = gs_wait(sessionid)`
  This is used to wait for the completion of an asynchronous call, and fetch the resulting output. On error, the output is all blank.
- `status = gs_cancel(sessionid)`
  This is used to wait for the completion of an asynchronous call, and fetch the resulting output. The status is 0 on success.
- `status = gs_get_last_error`
  Returns an error number for the last error that occurred.
- `str = gs_error_string(errnum)`
  Returns a string message for the error errnum.

5.4 Example Matlab session

The following example shows how the Matlab client can be used. The function that is called (`vpass_int`) simply sends an integer vector back and forth, doing nothing useful. It is used for testing and timing GridSolve.

```
>> gs_info('vpass_int')
Description of call:
<problem name="vpass_int" type="subroutine" description="Does nothing...just for testing perfo
 <arglist count="2">
  <arg name="ivec" inout="inout" datatype="int" objectype="vector" rowexp="n" colexp="1" descr
  <arg name="n" inout="in" datatype="int" objectype="scalar" rowexp="1" colexp="1" description
 </arglist>
 <infolist count="4">
  <info type="LANGUAGE" value="C" />
  <info type="LIBS" value="-L$(top_builddir)/problems/passing -lpass" />
  <info type="COMPLEXITY" value="1.0" />
  <info type="MAJOR" value="ROW" />
 </infolist>
</problem>
Matlab call prototype:
[ ivec  ] = vpass_int(ivec, n)
>> ivec = rand(10,1);
ivec = rand(10,1);
>> [sessionid] = gs_call_async('vpass_int', ivec, 10);
>> status = gs_probe(sessionid);
>> status
status =
    1
>> [outvec] = gs_wait(sessionid);
>> size(outvec)
ans =
    10     1
```

Chapter 6

GridSolve Request Farming

6.1 Introduction

Farming is a way of calling GridSolve to manage large numbers of requests for a single GridSolve problem. Many GridSolve users are confronted by situations when many somewhat similar computations must be performed in parallel. One way to do this in GridSolve is to write non-blocking calls to `grpc_call_async()` in C for instance. However, this can become cumbersome. In the present distribution, this call, `grpc_farm()`, is only available from C and Matlab. A Fortran interface will most likely not be provided because of pointer management.

6.2 Calling Farming in C

Like `grpc_call()` and `grpc_call_async()`, the `grpc_farm()` function takes a variable number of arguments. Its first argument is a string that describes the iteration range. This string is of the form `i=%d,%d` (in C string format symbols). The second argument is a problem name appended with an opening and a closing parenthesis. The arguments following are similar in intent to the ones supplied to `grpc_call()`, but are iterators as opposed to integers or pointers. Where the user was passing, say an integer, to `grpc_call()`, he now needs to pass an array of integers and tell `grpc_farm()` which element of this array is to be used for which iteration. This information is encapsulated in an iterator and we provide three functions to generate iterators:

```
grpc_int()
grpc_int_array()
grpc_ptr_array()
```

Let us review these functions one by one.

- grpc_int() – This function takes only one argument: a character string that contains an expression that is evaluated to an integer at each iteration. The format of that string is based on a Shell syntax. $i represents the current iteration index, and classic arithmetic operators are allowed. For instance:

  ```
  grpc_int("$i+1")
  ```

  returns an iterator that generates an integer equal to one plus the current iteration index at each iteration.

- grpc_int_array() – This function takes two arguments: i. a pointer to an integer array (int *); ii. a character string that contains an expression. For instance,

  ```
  grpc_int_array(ptr,"$i")
  ```

  returns an iterator that generates at each iteration an integer equal to the i^{th} element of the array ptr where i is the current iteration index.

- grpc_ptr_array() – This function takes two arguments: i. a pointer to an array of pointers (void **); ii. a character string that contains an expression. For instance,

  ```
  grpc_ptr_array(ptr,"$i")
  ```

  returns an iterator that generates at each iteration a pointer which is the i^{th} element of the array ptr where i is the current iteration index.

6.3 An example

Let us assume that the user wants to sort an array of integers with GridSolve using the C interface. The default GridSolve server comes with a default problem called iqsort that does a quicksort on an integer vector. The call looks like

```
status = grpc_call(&handle,size,ptr,sorted);
```

where size is the size of the array to be sorted, ptr is a pointer to the first element of the array, and sorted is a pointer to the memory space that will hold the sorted array on return. What if the user wants to sort 200 arrays? One way is to write 200 calls as the one above. Not only would it be tedious, but also inefficient as the sorts would be done successively, with no parallelism. In order to obtain parallelism, one must call `grpc_call_async()` and make the corresponding calls to `grpc_probe()` and `grpc_wait()` as explained in Chapter 4 or use `grpc_farm()`. Before calling `grpc_farm()`, the user needs to construct arrays of pointers and integers that contain the arguments of each of the GridSolve calls. This is straightforward: where the user would have called GridSolve as:

```
status1 = grpc_call_async(&handle, &request1, size1, ptr1, sorted1);
status2 = grpc_call_async(&handle, &request2, size2, ptr2, sorted2);
...
status200 = grpc_call_async(&handle, &request200, size200, array200, sorted200);
```

and then to have calls to `grpc_probe()` and `grpc_wait()` for each request. With farming, one only needs to construct three arrays as:

```
int size_array[200];
void *ptr_array[200];
void *sorted_array[200];
size_array[0] = size1;
ptr_array[0] = ptr1;
sorted_array[0] = sorted1;
...
```

Then, `grpc_farm()` can be called as:

```
status_array = grpc_farm("i=0,199",&handle,
grpc_int_array(size_array,"$i"),
grpc_ptr_array(ptr_array,"$i"),
grpc_ptr_array(sorted_array,"$i"));
```

In short, `grpc_farm()` is a concise, convenient way of farming out groups of requests. Of course, it uses `grpc_call_async()` underneath, thereby ensuring fault-tolerance and load-balancing.

6.4 Catching errors

`grpc_farm()` returns an integer array. That array is dynamically allocated and must be freed by the user after the call. The array is at least of size 1. The first element of the array is either `GRPC_NO_ERROR` or some GridRPC error code such as `GRPC_OTHER_ERROR_CODE`. If it is `GRPC_NO_ERROR`, then the call was completed successfully and the array is of size 1. If the first element of the array is not `GRPC_NO_ERROR`, then at least one of the requests failed. The array is then of size one plus the number of requests and the $(1+i)^{th}$ element of the array is the error code for the i^{th} request. Here is an example on how to print error messages:

```
status = grpc_farm("i=0,200",....);
if (status[0] == GRPC_NO_ERROR) {
  fprintf(stderr,"Success\n");
} else {
  for (i=1;i<201;i++) {
    fprintf(stderr,"Request #%d:",i);
    fprintf(stderr,"reason: %s\n", grpc_error_string(status[i]));
  }
}
free(status);
```

6.5 Farming in Matlab

TBA

Chapter 7

Running the GridSolve Agent

After compiling the agent as explained in Chapter 2, the executable of the GridSolve agent is:

```
$GRIDSOLVE_ROOT/src/agent/GS_agent
```

The proper command line for this program is

```
GS_agent [-c] [-l logfile] [-w httpd_port]
```

When invoked with no arguments, a stand-alone agent is started. This agent is now available for registrations of GridSolve servers wanting to participate in a new GridSolve system. After servers are registered, client programs can contact this agent and have requests serviced by one or more of the registered servers. If there is already an agent running on the machine, you will need to adjust the environment variables to avoid conflicts with the ports that are already in use. See Appendix A for details.

The `-l` option specifies the name of a file to use for logging purposes.

```
% GS_agent -l /home/user/agent_logfile
```

This file is where the agent logs all of its interactions (and possibly errors) since it is a daemon with no controlling terminal and therefore has no way to do this otherwise. This log file also produces very useful information about requests, among other things, that helps administrators know how their GridSolve system is being used. If no `-l` option is specified, the default log file is `$GRIDSOLVE_ROOT/gs_agent.log`. This means that successive runs of the agent with no specification of a log file will overwrite the original log file, so if the information is needed, it must be copied to another file. To terminate an existing agent (or query an existing GridSolve system), the user should refer to the GridSolve management tools, particularly `GS_killagent`, as outlined in Chapter 9.

If you do not want to run the agent as a daemon and would like to see all output logged to the console instead of a file, specify the `-c` option.

The -w option allows changing the port on which the agent's http daemon listens. By default, the daemon attempts to use port 8080. If “disable” is specified as the port, the agent will not attempt to start the http daemon.

Chapter 8

Running the GridSolve Server

After compiling the server as explained in Chapter 2, the executable of the GridSolve server is:

```
$GRIDSOLVE_ROOT/src/server/GS_server
```

The proper command line for this program is

```
GS_server [-c] [-l logfile] [-s server config]
```

This executable uses a configuration file for initializing the GridSolve server. The default configuration file is `$GRIDSOLVE_ROOT/server_config`. This is the file that should be used for first experiments and for testing the system. However, it is possible to customize or expand the functionality of a server by modifying this file. The `-s` option may be used to specify an alternate location for the file, for example:

```
% GS_server -s /tmp/test/server_config
```

The `-l` option specifies the name of a file to use for logging purposes.

```
% GS_server -l /home/user/server_logfile
```

This file is where the server logs all of its interactions (and possibly errors) since it is a daemon with no controlling terminal and therefore has no way to do this otherwise. This log file also produces very useful information about requests, among other things, that helps administrators know how their GridSolve system is being used. If no `-l` option is specified, the default log file is `$GRIDSOLVE_ROOT/gs_server.log`. This means that successive runs of the server with no specification of a log file will overwrite the original log file, so if the information is needed, it must be copied to another file. To terminate an existing server (or query an existing GridSolve system), the user should refer to the GridSolve management tools, particularly `GS_killserver`, as outlined in Chapter 9.

Note: When running multiple servers within the same directory tree, if a unique log file is not specified, then the most recently started server will take over the log file. Log messages from other servers will be lost. Use the `-l` parameter to specify a unique log for each server to avoid this.

If you do not want to run the server as a daemon and would like to see all output logged to the console instead of a file, specify the `-c` option.

8.1 The Server Configuration File

The server configuration file is used to customize the server. The default configuration file in `$GRIDSOLVE_ROOT/server_config` should be used as a template to create new configuration files. This configuration file is organized as follows. A line can contain one of three things:

- A comment – if the line starts with a # (pound symbol) then the remainder is ignored and may be used for comments.
- Nothing – if the line is blank, it is ignored.
- Attribute Assignment – these assignments take the form

  ```
  ATTRIBUTE=VALUE
  ```

  where `ATTRIBUTE` is the name of the attribute being defined and `VALUE` is a string representing the value to be assigned. For example

  ```
  AGENT=gridsolve.cs.utk.edu
  ```

Let us review some of the possible attributes and how they can be used to precisely define a GridSolve server as it is done in the default configuration file.

- `AGENT` – the name of the host running the agent
- `PORT` – the port on which this server should listen
- `OUTPUT_TTL` – the number of seconds to allow unretrieved results to remain stored on disk

In addition, you may define your own attributes. These will be reported to the agent upon registration of the server and may be used by the client for filtering the server selection.

8.2 Server Restrictions

Sometimes it is useful to restrict the circumstances under which a server will accept jobs. The GridSolve server supports two methods of restricting usage: by time and by the number of running jobs.

For example, to only accept jobs from 9am to 5pm (local time), add a line to the server_config file such as:

```
RESTRICT_TIME=9:00am-5:00pm
```

The beginning and ending times may formatted as “H:M:S”, “H:M”, or “H”. If “am” or “pm” is not appended the time is assumed to be in 24-hour format.

The server can also limit the total number of jobs that it will run at a time. For example, to allow only three jobs to run at a time, add a line to the server_config file such as:

```
RESTRICT_JOBS=3
```

8.3 Adding Services to a GridSolve Server

Before incorporating a function into GridSolve, the user must write a GridSolve Interface Definition Language (gsIDL) file that describes the calling sequence. See Chapter 10 for more detail on writing these files. Once the gsIDL file has been written, it must be compiled using the GridSolve problem compiler in the `$GRIDSOLVE_ROOT/src/problem` directory. For example:

```
% problem_compile ddot.idl
```

The problem compiler generates a service directory (in `$GRIDSOLVE_ROOT/service`) for each problem specification in the gsIDL file. In this service directory the problem compiler creates a service executable that is executed by the GridSolve server. Therefore, the server administrator does not need to restart the server to add a new service.

Chapter 9

GridSolve Management Tools for Administrators

The GridSolve distribution comes with tools to manage the GridSolve system. After compilation the following executables are available:

```
$GRIDSOLVE_ROOT/src/tools/GS_config
$GRIDSOLVE_ROOT/src/tools/GS_get_example
$GRIDSOLVE_ROOT/src/tools/GS_killagent
$GRIDSOLVE_ROOT/src/tools/GS_killserver
$GRIDSOLVE_ROOT/src/tools/GS_probdesc
$GRIDSOLVE_ROOT/src/tools/GS_problems
```

Let us review these executables one by one.

- `GS_config` – This executable takes one argument on the command line – the name of a host running a GridSolve agent. It then prints a list of servers participating in the GridSolve system:

  ```
  % GS_config cupid.cs.utk.edu
  AGENT: cupid [3 servers]
  SERVER: ig.cs.utk.edu (160.36.58.91:9000)
  SERVER: kiransagi (160.36.253.12:9000)
  SERVER: ns4 (192.168.0.5:9000, proxy=160.36.58.63:8888)
  ```

  For servers that are proxied, the proxy information is printed also.

- `GS_get_example` – This is used to request a C source code example for the specified service. The usage is as follows.

  ```
  Usage: GS_get_example <problem name> [server name]
  ```

  The name of the problem is required, but a specific server host name is optional. If specified, the example will be requested from that server. The C source code is then printed to stdout.

- `GS_killagent` – This executable takes one argument on its command line – the name of a host running a GridSolve agent. After a (basic) user authentication, the executable kills the agent.

  ```
  % GS_killagent gridsolve.cs.utk.edu
  ```

  For this beta release, the password to kill agents and servers is hardcoded to “GridSolve”, however in the first official release we will have authentication enabled for these tools.

- `GS_killserver` – This executable takes two arguments on its command line – the name of a host running a GridSolve agent and the name of a host running a GridSolve server. After a (basic) user authentication, the executable kills the server, using the agent as an entry-point into the system.

  ```
  % GS_killserver gridsolve.cs.utk.edu cupid.cs.utk.edu
  ```

- `GS_problems` – This executable takes one argument on the command line – the name of a host running a GridSolve agent. It then prints a list of problems that can be solved by contacting that agent.

  ```
  % GS_problems cupid.cs.utk.edu
  AGENT: cupid [26 problems]
  dgesv
  dposv
  ddot
  daxpy
  dgemv
  dgemm
  ctotal
  ftotal
  sleeptest
  ns_abort
  return_int_scalar
  return_float_scalar
  return_double_scalar
  return_char_scalar
  return_int_vector
  return_float_vector
  return_double_vector
  return_char_vector
  return_int_matrix
  return_float_matrix
  return_double_matrix
  return_char_matrix
  vpass_int
  mpass_int_rowmaj
  varlen_return
  mandel
  ```

- `GS_probdesc` – This executable takes two arguments on the command line. The first argument is the name of a host running a GridSolve agent and the second argument is the name of the problem whose description should be printed. It then prints a detailed description of the specified problem:

```
% GS_probdesc cupid.cs.utk.edu ddot

Problem Name: ddot

Problem Description:
  Forms the dot product of two vectors.
Double Precision routine.
http://www.netlib.org/blas/

Argument 0:
  Argument Name:     n
  Description:       none
  In/out mode:       in
  Data type:         int
  Object type:       scalar
  Row size expr:     1
  Column size expr:  1

Argument 1:
  Argument Name:     dx
  Description:       none
  In/out mode:       in
  Data type:         double
  Object type:       vector
  Row size expr:     n*incx
  Column size expr:  1

Argument 2:
  Argument Name:     incx
  Description:       none
  In/out mode:       in
  Data type:         int
  Object type:       scalar
  Row size expr:     1
  Column size expr:  1

Argument 3:
  Argument Name:     dy
  Description:       none
  In/out mode:       in
  Data type:         double
  Object type:       vector
  Row size expr:     n*incy
  Column size expr:  1

Argument 4:
  Argument Name:     incy
  Description:       none
  In/out mode:       in
  Data type:         int
  Object type:       scalar
  Row size expr:     1
  Column size expr:  1

Argument 5:
  Argument Name:     __retval
  Description:       Return value
  In/out mode:       out
  Data type:         double
  Object type:       scalar
  Row size expr:     1
  Column size expr:  1

Problem attributes:
  LANGUAGE: FORTRAN
  LIBS: $(BLAS_LIBS)
  COMPLEXITY: 2.0*N
  MAJOR: COLUMN
```

Chapter 10

GridSolve Interface Definition Language

The GridSolve Interface Definition Language (gsIDL) is the mechanism through which GridSolve enables services to be invoked on behalf of the user. GridSolve comes with several example gsIDL files in the `$GRIDSOLVE_ROOT/problems/idl` directory. First we will show a simple example and then examine the gsIDL file format in more detail.

10.1 gsIDL Example

Suppose we want to integrate the BLAS routine `ddot` (which computes the dot product of two vectors) into GridSolve. As you can see from the original Fortran header, it takes two vectors, a length argument, and a stride argument for each of the vectors:

```
double precision function ddot(n,dx,incx,dy,incy)
double precision dx(*),dy(*)
integer n,incx,incy
```

The gsIDL file corresponding to this function would be:

```
1   FUNCTION double ddot(IN int n, IN double dx[n*incx], IN int incx,
2     IN double dy[n*incy], IN int incy)
3   "Dot product (from BLAS)"
4   LANGUAGE = "FORTRAN"
5   LIBS = "/usr/local/lib/libf77blas.a /usr/local/lib/libatlas.a"
6   COMPLEXITY = "2.0*N"
7   MAJOR="COLUMN"
```

Now we examine this file line-by-line.

- Lines 1-2: This is the header, which defines the arguments that appear in the function to be called by GridSolve. It resembles the original function declaration, but GridSolve requires a bit of extra information. For each argument, it needs to know whether it is modified by the function. In this case, none of the arguments are modified, so we declare them all as `IN`, meaning input-only. The full range of possibilites will be explained in more detail later. For non-scalar arguments, we must also specify the size of the argument in terms of some scalar arguments. This can be a mathematical expression, as shown in this example. Since `n` is the number of elements (*not* the total vector length) and `incx` is the stride for `dx`, the total length of the `dx` vector that must be sent to the server is `n*incx`. Thus we declare the vectors as `dx[n*incx]` and `dy[n*incy]`.

- Line 3: This line is a string describing the function.
- Line 4: This line specifies the language in which the function is implemented.
- Line 5: This line specifies the libraries that need to be linked. In this case we link the ATLAS library since it contains the implementation of the `ddot` function that we want GridSolve to call.
- Line 6: This is an expression that specifies the asymptotic complexity (or *big-O* bounds) for the algorithm. It is expressed in terms of constants and/or arguments from the gsIDL function delcaration (lines 1-2). The typical mathematical operators are allowed, as explained in more detail in Section **??**.
- Line 7: This line specifies whether the algorithm is row-major or column-major. In this case it does not really matter since it is not a matrix algorithm. GridSolve will automatically transpose matrices when calling from a row-major client to a column-major service (or vice versa).

10.2 Description of the gsIDL Grammar

The EBNF grammar for the gsIDL file is:

Start	::=	IDL_PARSE Problemlist
Start	::=	EXPR_EVAL_TOK DimEvaluated
Identifier	::=	IDENTIFIER
Constant	::=	CONSTANT
StringLiteral	::=	STRING_LITERAL
Problemlist	::=	[Problemlist] Problem
Problem	::=	ProbSpec Identifier “**(**” Arglist “**)**” Description Infolist
ProbSpec	::=	FUNCTION Datatype “[” Dim “]” “[” Dim “]”
ProbSpec	::=	FUNCTION Datatype “[” Dim “]”
ProbSpec	::=	FUNCTION Datatype
ProbSpec	::=	SUBROUTINE
Infolist	::=	[Infolist] Info
Info	::=	Infotype “=” StringLiteral
Arglist	::=	[[Arglist “**,**”] Arg]
Arg	::=	Inout (Datatype Identifier “[” Dim “]” “[” Dim “]” “{” SpDim “**,**” SpDim “**,**” SpDim “}” Description | Datatype Identifier “[” Dim “]” “[” Dim “]” Description | Datatype Identifier “[” Dim “]” Description | Datatype Identifier Description | FILE_TOK Identifier Description | FILE_TOK Identifier “[” Dim “]” Description)
SpDim	::=	Identifier
Inout	::=	IN_TOK
Inout	::=	INOUT_TOK

Inout	::=	OUT_TOK
Inout	::=	VAROUT
Inout	::=	WORKSPACE
Description	::=	[StringLiteral]
Datatype	::=	INT_TOK
Datatype	::=	CHAR_TOK
Datatype	::=	FLOAT_TOK
Datatype	::=	SCOMPLEX
Datatype	::=	DCOMPLEX
Datatype	::=	DOUBLE_TOK
Infotype	::=	LANGUAGE
Infotype	::=	MAJOR
Infotype	::=	LIBS
Infotype	::=	INCLUDES
Infotype	::=	COMPLEXITY
Infotype	::=	PARALLEL
Infotype	::=	CODE
Infotype	::=	Identifier
DimEvaluated	::=	expression
Dim	::=	expression
primary_expression	::=	Identifier
primary_expression	::=	Constant
primary_expression	::=	“**(**” expression “**)**”
postfix_expression	::=	primary_expression
postfix_expression	::=	Identifier “**(**” “**)**”
postfix_expression	::=	Identifier “**(**” argument_expression_list “**)**”
argument_expression_list	::=	[argument_expression_list “**,**”] expression
unary_expression	::=	postfix_expression
unary_expression	::=	unary_operator cast_expression
unary_operator	::=	“+”
unary_operator	::=	“−”
unary_operator	::=	“ ”
unary_operator	::=	“!”
cast_expression	::=	unary_expression
cast_expression	::=	“**(**” type_specifier “**)**” cast_expression
multiplicative_expression	::=	[multiplicative_expression (“∗” | “/” | “%”)] cast_expression
additive_expression	::=	[additive_expression (“+” | “−”)] multiplicative_expression
shift_expression	::=	[shift_expression (LEFT_OP | RIGHT_OP)] additive_expression
relational_expression	::=	[relational_expression (“<” | “>” | LE_OP | GE_OP)] shift_expression
equality_expression	::=	[equality_expression (EQ_OP | NE_OP)] relational_expression
and_expression	::=	[and_expression “&”] equality_expression
exclusive_or_expression	::=	[exclusive_or_expression “ˆ”] and_expression
inclusive_or_expression	::=	[inclusive_or_expression “|”] exclusive_or_expression
logical_and_expression	::=	[logical_and_expression AND_OP] inclusive_or_expression

logical_or_expression	::=	[logical_or_expression OR_OP] logical_and_expression
expression	::=	logical_or_expression ["**?**" expression "**:**" expression]
type_specifier	::=	CHAR_TOK
type_specifier	::=	SHORT_TOK
type_specifier	::=	INT_TOK
type_specifier	::=	LONG_TOK
type_specifier	::=	FLOAT_TOK
type_specifier	::=	DOUBLE_TOK

As you can see from the grammar, each problem should begin with the problem specification followed by a string description. After that, the problem attributes (`LANGUAGE`, `MAJOR`, etc.) may be specified in any order.

In the grammar, `WORD` represents an identifier which begins with a letter and is followed by sequence of letters, digits, or underscores. It would be expressed as a regular expression as follows:

```
[a-zA-Z]([0-9]|[a-zA-Z]|_)*
```

`STR_CONST` is an arbitrary string enclosed with double quotes. All the other terminals in the grammar are keywords with the same name in the gsIDL.

Notice that each argument in *Arglist* is prefaced with an *Inout* specifier. This describes how the argument is to be passed to the server. the possible categories are:

- `IN` – input-only, allocated by the client and not modified by the function
- `OUT` – output-only, allocated by the client and initialized by the function
- `INOUT` – input-output, allocated and initialized by the client and modified by the function
- `VAROUT` – output-only, allocated and initialized by the function
- `WORKSPACE` – this is used to represent Fortran "workspace" arguments which you want to leave out of the client calling sequence. These will be allocated by the server and do not get transmitted over the wire.

Most of the *Infotype* keywords were described in the gsIDL example earlier. The others represented in the grammar are reserved for future use.

10.3 Determining the C Client Calling Sequence

In this section we will describe how to write the client code to call any gsIDL. The easiest way to understand the calling sequence for a given gsIDL is to compile it and look at the example client code that is generated by the GridSolve gsIDL compiler. It will be named `<PROB_NAME>_grpc_example.c`, where `<PROB_NAME>` is the name of the service.

In general, the client call will have one argument for each argument in the gsIDL problem specification. There are two exceptions.

1. If the argument is classified as `WORKSPACE`, then it is omitted from the client calling sequence.

2. If the problem is declared as a `FUNCTION` (as opposed to a `SUBROUTINE`, which has no return value), there will be an additional argument at the end of the normal client calling sequence to hold the return value. It is considered an output-only argument, so it should be passed by reference. This is done because the GridRPC calls return a status (or request ID for non-blocking calls), so they cannot also return the function's return value.

One of the main characteristics that is relevant to determining how an individual argument should be passed is whether the argument is scalar or non-scalar. If the argument is a scalar and is input-only, then it is passed by value. Otherwise it should be passed by reference. If the argument is non-scalar, then it is always passed by reference. One special case is `VAROUT` which allows the service to return a variable-length non-scalar. In this case, the argument should be passed as pointer-to-pointer.

10.4 Determining the Fortran Client Calling Sequence

The NetSolve compatibility layer contains a Fortran 77 API. Fortran differs from C in that all arguments are passed by reference. GridSolve will handle dereferencing the pointers for arguments that are expected to be passed by value, so you should just pass the arguments as normal from Fortran.

Chapter 11

Interfacing with Batch Queues

Some machines, typically large parallel machines or clusters, can only be used by submitting the job to a batch queue. To allow GridSolve to work on such machines, we need to provide support for batch queue submission. However, since there is a wide variety of batch queue systems, each with their own commands and interfaces, we wanted to allow this feature to be customizable by the administrator of the server to suit the specifics of their site.

We have defined three basic queue operations: submit, probe, and cancel. For each of these operations, a script must be written to the following specifications.

11.1 Submit Script

GridSolve will pass one argument to the submit script, which is the name of the batch executable to be run. You will probably need to pass this executable name to the batch queue submit command. Also within the script, you should pass one argument to the batch executable, which is the full path of the request directory. You should normally use `$PWD` since this script would be invoked by the service process which is already in the request directory, but the batch executable needs to know where to begin because after submission it may start in a different directory.

Whatever back-end system you submit to, this script should only produce one line on stdout: a job identifier that can be used by the probe and cancel scripts to check status and kill the job, respectively.

GridSolve submit scripts should exit with the appropriate status as follows:

- 0 – the job was successfully submitted
- non-zero – there was a failure submitting the job

11.2 Probe Script

GridSolve will pass one argument to the probe script, which is the identifier of the job to probe. This is the job identifier produced earlier by the submit script.

GridSolve probe scripts should exit with the appropriate status as follows:

- 0 – the job is still running
- 1 – the job has completed

- 2 – the job terminated abnormally

11.3 Cancel Script

Arguments: GridSolve will pass one argument to the cancel script, which is the identifier of the job to kill. This is the job identifier produced earlier by the submit script.

GridSolve cancel scripts should exit with the appropriate status as follows:

- 0 – the job was successfully killed
- 1 – failed to kill the job

11.4 Examples

11.4.1 gsIDL Specification

Before a service can be batch-enabled, the names of the submit, probe, and cancel scripts must be specified in the gsIDL for the service (and then recompile the service). An example gsIDL file follows.

```
SUBROUTINE batch_test_int(INOUT int x[n], IN int n, IN int delay)
"Sorts an array of integers."
LANGUAGE = "C"
LIBS = "-L$(top_builddir)/problems/sorting -lsorting"
COMPLEXITY = "n"
MAJOR="ROW"
BATCH_SUBMIT="$(top_builddir)/examples/batch_scripts/gs_dummy_submit"
BATCH_PROBE="$(top_builddir)/examples/batch_scripts/gs_dummy_probe"
BATCH_CANCEL="$(top_builddir)/examples/batch_scripts/gs_dummy_cancel"
```

As you can see from this example, the batch scripts are specified in the service attribute section of the gsIDL file. Aside from those attributes, the file does not need to be modified.

11.4.2 Example Submit Script

In this example, you can see that the batch system requires a special script instead of a binary. So, in this submit script, we create the batch script with some default values. In this case the actual submit command prints only the job identifier, so we do not need to parse the output.

```
TMP_SCRIPT=gs_tmp_script
/bin/rm -f ${TMP_SCRIPT}

cat << EOF > ${TMP_SCRIPT}
#!/bin/bash
#PBS -l nodes=1:ppn=2
#PBS -l walltime=01:00:00
```

```
foo=\‘cat \$PBS_NODEFILE | awk -F: ’{print \$1}’\‘
ssh \${foo} $1 $PWD
EOF

qsub ${TMP_SCRIPT}
```

11.4.3 Example Probe Script

In this example, the batch queue has a command `tracejob` to get the status of a previously submitted job. We use this information to determine the proper exit status.

```
TRACEJOB=‘which tracejob‘

if qstat $1 >& /dev/null; then
  exit 0
else
  if [ "${TRACEJOB}" = "" ]; then
    exit 1
  else
    exit_status=‘tracejob $1 | egrep Exit_status | cut -d ’=’ -f 2‘
    if [ "${exit_status}" = "0" ]; then
      exit 1
    else
      exit 2
    fi
  fi
fi
```

11.4.4 Example Cancel Script

Cancelling a job is simple since the batch queue system has command to do it. We just need to make sure to exit with the appopriate status.

```
if qdel $1 >& /dev/null; then
  exit 0
else
  exit 1
fi
```

Chapter 12

Distributed Storage Infrastructure (DSI) in GridSolve

12.1 DSI Introduction

The Distributed Storage Infrastructure (DSI) is an attempt towards achieving coscheduling of the computation and data movement over the GridSolve system. The DSI API helps the user in controlling the placement of data that will be accessed by a GridSolve service. This is useful in situations where a given service accesses a single block of data a number of times. Instead of multiple transmissions of the same data from the client to the server, the DSI feature helps to transfer the data from the client to a storage server just once, and relatively cheap multiple transmissions from the storage server to the computational server. Thus the present DSI feature helps GridSolve to operate in a cache-like setting. Presently, only Internet Backplane Protocol (IBP) is used for providing the storage service. In the future, we hope to integrate other commonly available storage service systems.

12.2 Using DSI

To use DSI, one should enable the DSI feature both at the GridSolve client and the server. Type

```
% ./configure --with-dsi-ibp=IBP_DIR
```

during the initial configure of GridSolve. Here IBP_DIR denotes the location of the IBP directory. This is specifically the directory of the IBP full distribution downloadable from the IBP web site `http://loci.cs.utk.edu/ibp/`. Note: When using IBP in a server pool that has both IBP enabled servers and those that are not IBP enabled, one should use the assigned server feature to ensure that the problem submission goes to a server with IBP enabled.

12.3 DSI API

The DSI API is modeled after the UNIX file manipulation commands (open, close etc.) with a few extra parameters that are specific to the concepts of DSI. This section provides the syntax and semantics of the different DSI calls available to the GridSolve user.

12.3.1 grpc_dsi_open

This function is used for allocating a chunk of storage in the IBP storage.

```
grpc_error_t grpc_dsi_open(DSI_FILE **rfile, char* host_name, int flag,
                           int permissions, int size, dsi_type storage_system);
```

Parameters:

- `rfile` – Upon return, contains a pointer to the DSI file.
- `host_name` – Name of the host where the IBP server resides.
- `flag` – This flag has the same meaning as the flag in open() calls in C. Specifically `O_CREAT` is used for creating a DSI file.
- `permissions` – While creating the file with `O_CREAT` flag, the user can specify the permissions for himself and others. The permissions are similar to the ones used in UNIX. Hence if the user wants to set read and write permissions for himself and only read permissions for others, he would call grpc_dsi_open with 644 as the value for the permissions.
- `size` – Represents the maximum length of the DSI file. Write or read operations over this size limit will return an error.
- `storage_system` – At present, only IBP is supported.

On success, returns `GRPC_NO_ERROR`. On failure, returns `GRPC_OTHER_ERROR_CODE` as the major error code with one of the following minor error codes.

- `GRPC_DSI_UNKNOWN_FILE` – If the file does not exist and if the file is opened without `O_CREAT`.
- `GRPC_DSI_ALLOCATE_ERROR` – Error while allocating IBP storage.
- `GRPC_DSI_DISABLED` – If DSI is not enabled in the GridSolve configuration.

12.3.2 grpc_dsi_close

This function is used for closing a DSI file.

```
grpc_error_t grpc_dsi_close(DSI_FILE* dsi_file);
```

Parameters:

- `dsi_file` – Pointer to the DSI file.

On success returns `GRPC_NO_ERROR`. On failure, returns `GRPC_OTHER_ERROR_CODE` as the major error code with one of the following minor error codes.

- `GRPC_DSI_MANAGE_ERROR` – Error in IBP internals while closing.
- `GRPC_DSI_DISABLED` – If DSI is not enabled in the GridSolve configuration.

12.3.3 grpc_dsi_write_vector

This function is used for writing a vector of a particular datatype to a DSI file.

```
grpc_error_t grpc_dsi_write_vector(DSI_OBJECT **robject, DSI_FILE* dsi_file,
                                   void* data, int count, int data_type);
```

Parameters:

- `robject` – Upon return contains a pointer to the DSI object created for the vector.
- `dsi_file` – The name of the DSI file where the vector will be written.
- `data` – Vector to write to the DSI storage.
- `count` – Number of elements in the vector.
- `data_type` – One of GridSolve data types.

On success returns `GRPC_NO_ERROR`. On failure, returns `GRPC_OTHER_ERROR_CODE` as the major error code with one of the following minor error codes.

- `GRPC_DSI_STORE_ERROR` – Error while storing the vector in IBP.
- `GRPC_DSI_EACCES` – Not enough permissions for writing to the DSI file.
- `GRPC_DSI_DISABLED` – If DSI is not enabled in the GridSolve configuration.

12.3.4 grpc_dsi_write_matrix

Same functionality and return values as grpc_dsi_write_vector() except this function is used to write matrix of *rows* rows and *cols* columns.

```
grpc_error_t grpc_dsi_write_matrix(DSI_OBJECT **robject, DSI_FILE* dsi_file, void* data,
                                   int rows, int cols, int data_type);
```

12.3.5 grpc_dsi_read_vector

This function is used to read a vector of *count* items.

```
grpc_error_t grpc_dsi_read_vector(DSI_OBJECT* dsi_obj, void* data, int count,
                                  int data_type, int *bytes_read);
```

Parameters:

- `dsi_obj` – Pointer to the DSI object that contains the data to read.
- `data` – Actual vector to read.
- `count` – Number of elements of the vector to read.
- `data_type` – One of NetSolve data types.

- `bytes_read` – Upon return, contains the number of bytes read.

On success returns `GRPC_NO_ERROR`. On failure, returns `GRPC_OTHER_ERROR_CODE` as the major error code with one of the following minor error codes.

- `GRPC_DSI_LOAD_ERROR` – Error while loading the vector from IBP.
- `GRPC_DSI_EACCES` – Not enough permissions for reading from the DSI file.
- `GRPC_DSI_DISABLED` – If DSI is not enabled in the GridSolve configuration.

12.3.6 grpc_dsi_read_matrix

Same functionality and return values as grpc_dsi_read_vector() except grpc_dsi_read_matrix() is used to read matrix of *rows* rows and *cols* columns.

```
grpc_error_t grpc_dsi_read_matrix(DSI_OBJECT* dsi_obj, void* data, int rows, int cols,
                                  int data_type, int *bytes_read);
```

12.4 DSI Example

This section shows two example programs. Both programs call *int_vector_add5*, which adds 5 to every element of the input vector and stores the result into the output vector. The first example shows a standard call and the second example shows the DSI enabled version.

12.4.1 Standard Example

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

#include "grpc.h"

int
main(int argc, char *argv[])
{
  int int_vec_in[] = {93, 120, 84, 57, 147, 138, 66, 12, 88, 2};
  int *int_vec_out, i, n;
  grpc_function_handle_t handle;
  grpc_error_t status;

  n = sizeof(int_vec_in) / sizeof(*int_vec_in);

  int_vec_out = (int *)malloc(n * sizeof(int));

  if(grpc_initialize(NULL) != GRPC_NO_ERROR) {
    grpc_perror("grpc_initialize");
    exit(EXIT_FAILURE);
  }
  if(grpc_function_handle_default(&handle, "int_vector_add5") != GRPC_NO_ERROR) {
    fprintf(stderr,"Error creating function handle\n");
    exit(EXIT_FAILURE);
  }

  status = grpc_call(&handle, n, int_vec_in, int_vec_out);

  if(status != GRPC_NO_ERROR) {
    printf("GRPC error status  = %d\n", status);
    grpc_perror("grpc_call");
    exit(status);
  }

  for(i=0; i < n; i++) {
    if(int_vec_in[i] != int_vec_out[i] - 5) {
      fprintf(stderr, "Bad results in integer list\n");
      exit(EXIT_FAILURE);
    }
  }

  grpc_finalize();

  printf("Test successful\n");
  exit(EXIT_SUCCESS);
}
```

12.4.2 DSI Example

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

#include "grpc.h"

int
main(int argc, char *argv[])
{
  int int_vec_in[] = {93, 120, 84, 57, 147, 138, 66, 12, 88, 2};
  int *int_vec_out, i, n;
  grpc_function_handle_t handle;
  grpc_error_t status;
  DSI_OBJECT *int_vec;
  DSI_FILE *dsi_file;

  n = sizeof(int_vec_in) / sizeof(*int_vec_in);

  int_vec_out = (int *)malloc(n * sizeof(int));
  if(grpc_initialize(NULL) != GRPC_NO_ERROR) {
    grpc_perror("grpc_initialize");
    exit(EXIT_FAILURE);
  }

  if(grpc_dsi_open(&dsi_file, "localhost", O_CREAT|O_RDWR, 644, 30000, GS_DSI_IBP)
       != GRPC_NO_ERROR)
  {
    fprintf(stderr, "Error opening DSI file.\n");
    exit(EXIT_FAILURE);
  }

  if(grpc_dsi_write_vector(&int_vec, dsi_file, int_vec_in, n, GS_INT)
       != GRPC_NO_ERROR)
  {
    fprintf(stderr, "Error writing in_vec to DSI file.\n");
    exit(EXIT_FAILURE);
  }

  if(grpc_function_handle_default(&handle, "int_vector_add5") != GRPC_NO_ERROR) {
    fprintf(stderr,"Error creating function handle\n");
    exit(EXIT_FAILURE);
  }

  status = grpc_call(&handle, n, int_vec, int_vec_out);

  if(status != GRPC_NO_ERROR) {
    printf("GRPC error status  = %d\n", status);
    grpc_perror("grpc_call");
    exit(status);
  }

  for(i=0; i < n; i++) {
    if(int_vec_in[i] != int_vec_out[i] - 5) {
      fprintf(stderr, "Bad results in integer list\n");
      exit(EXIT_FAILURE);
    }
  }

  grpc_dsi_close(dsi_file);
  grpc_finalize();

  printf("Test successful\n");
  exit(EXIT_SUCCESS);
}
```

Chapter 13

GridSolve Profiling Interface

13.1 Introduction

The profiling interface is a very simple mechanism for providing specific timing information about the various aspects of a complete job submission. We developed this to be used internally to compare GridSolve with NetSolve, but it may be of some interest to end users as well. Since the NetSolve and GridSolve versions use the same fields, some of them may not be relevant to both systems, so such fields will always show an elapsed time of 0 in GridSolve.

13.2 Using the Profiling Interface

To use the profiling interface, first declare a variable of type `grpc_profile_t`. This structure should be passed to `grpc_profile()` before using any of the GridRPC call functions. When making several non-blocking calls, make sure not to pass the same structure to `grpc_profile()` or the timing information from different calls will be overwritten.

```
grpc_error_t grpc_profile(grpc_profile_t *prof)
```

If successful, this function returns `GRPC_NO_ERROR`. On failure, it will return

- `GRPC_NOT_INITIALIZED` – if GridRPC isn't initialized yet.
- `GRPC_OTHER_ERROR_CODE` (with minor errno: `GRPC_PROFILING_NOT_ENABLED`) if profiling was not enabled during configuration (see Section 2.1).

After the service has completed and the results have been retrieved, the profiling information can be accessed. The available fields, which are all double precision floating point values, follow below.

- `proxy_start` – unused in GridSolve
- `object_init` – unused in GridSolve
- `agent_comm` – the time to contact the agent and retrieve the server list
- `send_input` – the time to send all the input data
- `job_complete` – unused in GridSolve
- `recv_output` – the time to receive the output data

13.3 Example

```
...
grpc_profile_t gsprof;
...

prof_enabled = grpc_profile(&gsprof) == GRPC_NO_ERROR;

status = grpc_call(&handle, x, i);

if(prof_enabled)
  printf("%d: %g %g %g %g %g %g\n",(int)(i*sizeof(x[0])),
    gsprof.proxy_start, gsprof.object_init, gsprof.agent_comm,
    gsprof.send_input, gsprof.job_complete, gsprof.recv_output);
```

Chapter 14

Using the NAT Proxy

As the rapid growth of the Internet began depleting the supply of IP addresses, it became evident that some immediate action would be required to avoid complete IP address depletion. The IP Network Address Translator [EF94] is a short-term solution to this problem. Network Address Translation allows reuse of the same IP addresses on different subnets, thus reducing the overall need for unique IP addresses.

As beneficial as NATs may be in alleviating the demand for IP addresses, they pose many significant problems to developers of distributed applications such as GridSolve [Moo02]. Some of the problems as they pertain to GridSolve include the following:

- IP addresses are not unique – In the presence of a NAT, a given IP address may not be globally unique. Typically the addresses used behind the NAT are from one of several blocks of IP addresses reserved for use in private networks, though this is not strictly required. Consequently any system that assumes that an IP address can serve as the unique identifier for a component will encounter problems when used in conjunction with a NAT.

- IP address-to-host bindings may not be stable – This has similar consequences to the first issue in that GridSolve can no longer assume that a given IP address corresponds uniquely to a certain component. This is because, among other reasons, the NAT may change the mappings.

- Hosts behind the NAT may not be contactable from outside – This currently prevents all GridSolve components from existing behind a NAT because they must all be capable of accepting incoming connections.

- NATs may increase network failures – This implies that GridSolve needs more sophisticated fault tolerance mechanisms to cope with the increased frequency of failures in a NAT environment.

To address these issues we have developed a NAT-tolerant communications framework for GridSolve. To avoid problems related to potential duplication of IP addresses, the GridSolve components will be identified by a globally unique identifier, in this case a 64-bit random number. In a sense, the component identifier is a network address that is layered on top of the real network address such that a component identifier is sufficient to uniquely identify and locate any GridSolve component, even if the real network addresses are not unique. This is somewhat similar to a machine having an IP address layered on top of its MAC address in that the protocol to obtain the MAC address corresponding to a given IP address is abstracted in a lower layer.

An important aspect to making this new communications model work is the *proxy*, which is a component that will allow servers to exist behind a NAT. Since a server cannot accept unsolicited connections from outside the private network, it must first register with a proxy. The proxy acts on behalf of the component behind the NAT by accepting incoming connections destined for it. The component behind the NAT keeps the connection with the proxy open as long as possible since it can only be contacted by other components while it has a control connection established with the proxy. To maintain good performance, the proxy only examines the header of the connection establishment message and uses a simple table-based lookup to determine where to forward the connection. Furthermore, to prevent the proxy from being abused, authentication can be enforced.

Since NATs may introduce more frequent network failures, we have implemented a protocol to allow GridSolve components to reconnect to the system and retrieve the results later. This allows the servers to store the results of a computation to be retrieved at some time later when the network problem has been resolved. Additionally, this would allow a client to submit a problem, break the connection, and reconnect later at a more convenient time to retrieve the results, even perhaps from a different machine than the one used to submit the problem.

14.1 Starting the NAT Proxy and Proxied Server

The NAT Proxy may be started anywhere on the accessible side of the NAT. By *accessible*, we mean that a client should be able to establish a connection to the proxy. The client may still have to go through a NAT on its side, but that is fine as long as it is going through the outbound direction. The NAT proxy is located in the `$GRIDSOLVE_ROOT/src/proxy` directory. To start it, simply execute the following command:

```
% proxy_server
```

Once the proxy has been started, you may start the server that exists behind the NAT. Since the server needs to request that the proxy handle incoming connections, you need to specify the location of the proxy before starting the server:

```
% setenv GRIDSOLVE_PROXY foo.cs.utk.edu:8888
```

The other components do not need any modification to communicate via the proxy.

Bibliography

[AAB$^+$02] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[EF94] K. Egevang and P. Francis. The IP Network Address Translator (NAT). RFC 1631, May 1994.

[Moo02] K. Moore. Recommendations for the Design and Implementation of NAT-Tolerant Applications. Internet-draft, February 2002. Work in Progress.

[TMW92] Inc. The Math Works. *MATLAB Reference Guide*. 1992.

[Wol96] S. Wolfram. *The Mathematica Book, Third Edition*. Wolfram Median, Inc. and Cambridge University Press, 1996.

Appendix A

Environment Variables

Table A.1 has a summary of the environment variables used by GridSolve, the components to which they are relevant, and the default value used if not set. More detailed descriptions appear after the table.

Environment Variable	Relevant To	Default
`GRIDSOLVE_AGENT_PORT`	Client, Server, Agent	9876
`GRIDSOLVE_AGENT`	Client, Server	none
`GRIDSOLVE_PROXY`	Client, Server	none
`GRIDSOLVE_ROOT`	Server	path detected during configure
`GRIDSOLVE_ARCH`	Server	arch string detected during configure
`GRIDSOLVE_HTTPD_PORT`	Agent	8080
`GRIDSOLVE_SENSOR_PORT`	Agent	9988
`GRIDSOLVE_SERVER_PORT`	Agent	9000
`GRIDSOLVE_KEYTAB`	Proxy	none
`GRIDSOLVE_USERS`	Proxy	none
`PROXY_LISTEN_PORT`	Proxy	8888

Table A.1: Summary of GridSolve Environment Variables

- `GRIDSOLVE_AGENT_PORT` – tells the agent the port on which it should listen and tells the client or server the port on which it should try to contact the agent.
- `GRIDSOLVE_AGENT` – the host name of the GridSolve agent.
- `GRIDSOLVE_PROXY` – the host name and port of the proxy server. For example, "gridsolve.cs.utk.edu:8888".
- `GRIDSOLVE_ROOT` – the full path to the root of the GridSolve installation. This normally does not need to be set since it can be determined during configuration. If you want to run out of a different directory than the code was built, you must set this environment variable.
- `GRIDSOLVE_ARCH` – the specification string for this architecture. This normally does not need to be set since it can be determined during configuration.
- `GRIDSOLVE_HTTPD_PORT` – the port on which the HTTP daemon should listen.

- `GRIDSOLVE_SENSOR_PORT` – the port on which the monitoring sensor should listen.
- `GRIDSOLVE_SERVER_PORT` – the port on which the server should listen.
- `GRIDSOLVE_KEYTAB` – name of the file containing the GridSolve service principal. This is used for Kerberos authentication to the proxy.
- `GRIDSOLVE_USERS` – name of the file containing the authorized user list. This is used for Kerberos authentication to the proxy.
- `PROXY_LISTEN_PORT` – the port on which the proxy should listen.

Appendix B

GridRPC API Specification

B.0.1 Initializing and Finalizing Functions

```
grpc_error_t grpc_initialize( char * config_file_name);
grpc_error_t grpc_finalize();
```

B.0.2 Remote Function Handle Management Functions

```
grpc_error_t grpc_function_handle_default(grpc_function_handle_t * handle,
       char * func_name);
grpc_error_t grpc_function_handle_init(grpc_function_handle_t * handle,
       char * host_name, char * func_name);
grpc_error_t grpc_function_handle_destruct(grpc_function_handle_t * handle);
grpc_error_t grpc_get_handle(grpc_function_handle_t **handle, int sessionId);
```

B.0.3 GridRPC Call Functions

```
grpc_error_t grpc_call(grpc_function_handle_t *handle, ...);
grpc_error_t grpc_call_async(grpc_function_handle_t *handle,
       grpc_sessionid_t *, ...);
```

B.0.4 Asynchronous GridRPC Control Functions

```
grpc_error_t grpc_probe(int sessionID);
grpc_error_t grpc_probe_or(grpc_sessionid_t *idArray, size_t length,
    grpc_sessionid_t *idPtr);
grpc_error_t grpc_cancel(int sessionID);
grpc_error_t grpc_cancel_all(void);
```

B.0.5 Asynchronous GridRPC Wait Functions

```
grpc_error_t grpc_wait(grpc_sessionid_t sessionID);
grpc_error_t grpc_wait_and(grpc_sessionid_t *idArray, size_t length);
grpc_error_t grpc_wait_or(grpc_sessionid_t *idArray, size_t length,
    grpc_sessionid_t *idPtr);
grpc_error_t grpc_wait_all(void);
grpc_error_t grpc_wait_any(grpc_sessionid_t *idPtr);
```

B.0.6 Error Reporting Functions

```
char * grpc_error_string(grpc_error_t error_code);
grpc_error_t    grpc_get_error(grpc_sessionid_t sessionID);
grpc_error_t    grpc_get_failed_sessionid(grpc_sessionid_t *sessionID);
```

Appendix C

NetSolve Compatibility

GridSolve is designed as a replacement for NetSolve, but at the time of this release, there are several NetSolve features that have not been implemented in GridSolve yet. At the same time, GridSolve offers several enhancements not found in NetSolve. In this appendix we outline these incompatibilities and enhancements.

C.1 Incompatibilites

- *API* – GridSolve does not include the sequencing API.
- *Backend* – Support for different Grid services such as Globus, Condor, and LFC has not been implemented as part of GridSolve, but nothing prevents you from writing a wrapper that calls whatever you want.
- *Clients* – Mathematica, Octave, and Excel interfaces are not supported in GridSolve.

C.2 GridSolve Enhancements

- *NAT Tolerance* – GridSovle includes a NAT proxy that can allow servers to run behind a NAT. The original NetSolve client protocol has been modified so that clients can easily run behind NATs (without requiring a proxy).
- *Performance* – Instead of XDR, GridSolve uses a *Receiver Makes Right* protocol for data transfer. This requires data conversion only on the receiving end. Also we have incorporated a more efficient matrix transpose routine for C to Fortran calling (or vice versa). GridSolve also provides a faster return from non-blocking calls by forking a separate process to handle the transmission of the input data.
- *Disconnect* – For very long running jobs, GridSolve provides the option to disconnect from the server and pick the results up later, even from a different machine.
- *IDL* – The language for specifying the calling sequence of a routine to be integrated into GridSolve has been streamlined. We provide the *workspace* argument type, which specifies that the server should allocate memory for the routine, but it does not need to be transferred over the wire. We provide the *varout* argument type, which allows variable-length output

arguments to be returned by the service routine. We allow arbitrary mathematical expressions to be used to specify the sizes of non-scalar arguments and to specify the complexity of the algorithm.

- *Server* – Services are compiled to statically-linked executables, so there are no issues with library paths or various flags for different linkers. The services are not linked in with the server binary itself, so to add a new service just requires building the new service and placing it in the proper subdirectory. The server does not need to be restarted to enable the new problem.
- *Client Criteria* – To allow filtering the list of servers returned by the agent, the client can specify the criteria that it wants satisfied. The criteria can be specified as a boolean expression (e.g. `MEMORY > 1024`).