GWD-R
Category: Recommendation
GridRPC Working Group

H. Nakada, NIAIST
S. Matsuoka, Tokyo Institute of Tech.
K. Seymour, Univ. of Tenn., Knoxville
J. Dongarra, Univ. of Tenn., Knoxville
C. Lee, The Aerospace Corp.
H. Casanova, UCSD, SDSC
November 12, 2003

A GridRPC Model and API

Abstract

This document presents a model and API for GridRPC, i.e., a remote procedure call (RPC) mechanism for grid environments. As a Recommendations track document in the Global Grid Forum, the goal of this document is to clearly and unambiguously define the syntax and semantics for GridRPC, thereby enabling a growing user base to take advantage of multiple implementations. The motivation for this document is to provide an easy avenue of adoption for grid computing, since (1) RPC is an established distributed computing paradigm, and (2) there is a growing user-base for network-enabled services. By doing so, this document will also facilitate the development of multiple implementations.

1. Introduction

The goal of this document is to clearly and unambiguously define the syntax and semantics for GridRPC, a remote procedure call (RPC) mechanism for grid environments, thereby providing an avenue of easy access to grid computing. As such, it is outside the scope of this document to review or discuss the important issues of network-enabled services or to provide any kind of tutorial information. Nonetheless, a Related Work section is provided to capture many references and pointers to relevant works that have lead up to this document. A preliminary version of this model and API appeared as [16]. A longer version of that paper is available as [17].

2. The Basic GridRPC Model

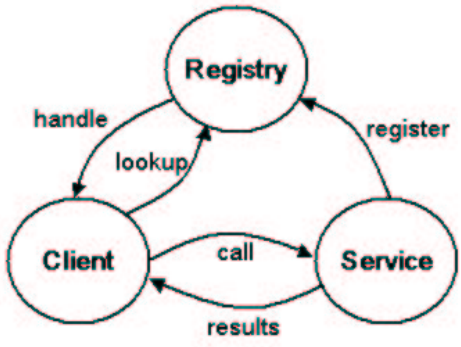

Figure 1. The Basic GridRPC Model.

Figure 1 illustrates the basic GridRPC model. The functions shown here are very fundamental and, hence, appear in many other systems. A service registers with a registry. A client subsequently contacts the registry to look-up a desired service and the registry returns a handle to the client. The client then uses the handle to call the service which eventually returns the results.

In the GridRPC terminology adopted here, the service handle is a *function handle* which represents a mapping from a simple, flat function name string to an instance of that function on a particular server. 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 GridRPC 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. Document Scope

This simple, common model nonetheless represents multiple fundamental issues. It is clearly impossible to deal with them all at the same time. Hence, we now clarify what this document defines and does not define.

3.1 In Scope

This document focuses on just defining the API and the minimal programming model needed to understand and use the API. More specifically, it focuses on simple, client-server interaction since this comprises the majority of usage scenarios.

3.2 Out of Scope

The following topics are very important but are nonetheless out of the scope of this document:

- **Service Discovery.** How the actual service registry or look-up is done is not addressed in this document. It is assumed that some type of registry or grid information service is available to accomplish this function.
- **Non-flat Service Names.** The current API assumes simple name strings for GridRPC services. Describing and discovering GridRPC services by attributes or metadata schemas would certainly be very useful but is not addressed here.
- **General Workflow.** General mechanisms for managing grid workflows are not in the scope of this document. Simple extensions to the API may be possible, however, that facilitate workflow management.
- **Interoperability between Implementations.** Since this document focuses on the GridRPC API, it says nothing about the protocols used to communicate between clients, servers, and registries. Hence, it does not address interoperability.

4. The GridRPC API

We begin by introducing the data types defined by GridRPC.

4.1 GridRPC Data Types

grpc_function_handle_t

Variables of this data type represent a specific remote function that has been bound to a specific server. They are allocated by the user. After a *function handle* is initialized, it may be used to invoke the associated remote function as many times as desired. The lifetime of a *function handle* is determined when the user invalidates the *function handle* with a *handle destruct* call.

grpc_sessionid_t

Variables of this data type represent a specific non-blocking GridRPC call. *Session IDs* are used to probe or wait for call completion, to cancel a call, or to check the error status of a call. *Session IDs* are also allocated by the user but their lifetime is determined automatically. A *session ID* is initialized when a non-blocking GridRPC call is made. It is invalidated, or destroyed, when (1) all return arguments have been received, and (2) a wait function has returned a "call complete" status to the application. If an invalid *session ID* is passed to any GridRPC call, an error will result.

grpc_arg_stack_t

This data type is used for argument stacks. (See Subsections 4.4 and 4.5.)

grpc_error_t

This data type is used for all error and return status codes from GridRPC functions.

4.2 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_error_t grpc_initialize(char *config_file_name)

This function reads the configuration file and initializes the required modules.

grpc_error_t grpc_finalize(void)

This function releases any resources being used by GridRPC.

4.3 Remote Function Handle Management Functions

The *function handle management* group of functions allows the creation and destruction of function handles.

grpc_error_t grpc_function_handle_default(
grpc_function_handle_t *handle,
char *func_name)

This creates a new function handle using a default server associated with the given function name. This default could be a pre-determined server or it could be a server that is dynamically chosen by the resource discovery mechanisms of the underlying GridRPC implementation.

grpc_error_t grpc_function_handle_init(
grpc_function_handle_t *handle,
char *host_port_str,
char *func_name)

This creates a new function handle with a server explicitly specified by the user. This explicit server is specified by a string of the form *"host_name:port_number"*.

grpc_error_t grpc_function_handle_destruct(grpc_function_handle_t *handle)

This releases all information and resources associated with the specified function handle.

grpc_error_t grpc_get_handle(grpc_function_handle_t *handle, grpc_sessionid_t sessionId)

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

4.4 GridRPC Call Functions

The four GridRPC call functions may be categorized by a combination of two properties: blocking behavior and calling sequence. A call may be either blocking (synchronous) or non-blocking (asynchronous) and it may use either a variable number of arguments (like *printf()*) or an *argument stack* calling sequence. The argument stack calling sequence allows building the list of arguments to the function at runtime through elementary stack operations, such as *push* and *pop*. Table 1 shows the appropriate function to use for each combination of the two call properties.

	Blocking	Non-blocking
Variable Argument List	**grpc_call()**	**grpc_call_async()**
Argument Stack	**grpc_call_arg_stack()**	**grpc_call_arg_stack_async()**

Table 1. GridRPC Call Functions

Rationale:
The use of stack arguments allows the GridRPC API to be used in middleware when the number of arguments is not known at compile-time. Stack arguments can be seen as more fundamental since all remote execution calls could be expressed as *apply(func_name, arg_stack)*. This can enable type-checking and is more portable than varargs. Allowing a variable number of arguments is, nonetheless, a tremendous convenience for end-users and a very common practice.
End of Rationale.

grpc_error_t grpc_call(grpc_function_handle_t *handle, <varargs>)

This makes a blocking remote procedure call with a variable number of arguments.

grpc_error_t grpc_call_async(
grpc_function_handle_t *handle,

grpc_sessionid_t *sessionID,
<varargs>)

This makes a non-blocking remote procedure call with a variable number of arguments. A *session ID* is returned that can be used to probe or wait for completion, cancel the call, and check for the error status of a call.

grpc_error_t grpc_call_arg_stack(grpc_function_handle_t *handle, grpc_arg_stack_t *args)

This makes a blocking call using the argument stack.

grpc_error_t grpc_call_arg_stack_async(
grpc_function_handle_t *handle,
grpc_sessionid_t *sessionID,
grpc_arg_stack_t *args)

This makes a non-blocking call using the argument stack. Similarly, a *session ID* is returned that can be used to probe or wait for completion, cancel the call, and check for the error status of a call.

The GridRPC Recommendation does not define which implementation-related operations may be assumed to be complete when an asynchronous call returns. However, all asynchronous GridRPC calls must return as soon as possible after it is safe for a user to modify any input argument buffers.

Rationale:
By returning as soon as possible, e.g., before the remote operation has started and before any results are returned, the GridRPC user can overlap the remote computation with other local computation. By allowing the user to modify any buffers after the asynchronous call returns, we present the user with the safest and simpliest buffer handling semantics possible. While it may be possible to improve performance further by allowing asynchronous calls to return before it is safe to modify input argument buffers, it was considered not worth the added complexity and "danger" for handling buffers. This is the approach taken by current prototype implementations.
End of Rationale.

4.5 Argument Stack Functions

These functions are used to construct a stack of arguments at runtime. When interpreted as a list of arguments, the stack is ordered from bottom up. That is, to emulate a function call **f(a,b,c)**, the user would push the arguments in the same order: **push(a); push(b); push(c);**. Note that an argument stack has a fixed size when it is allocated. It does not, however, have to be "fully populated" when used as a call argument. Only those arguments on the stack at the time of the call will be used.

grpc_error_t *grpc_stack_init(grpc_arg_stack_t *stack, int maxsize)

This initializes a new argument stack. *maxsize* is the maximum number of arguments that can be pushed on to this stack.

grpc_error_t grpc_stack_push(grpc_arg_stack_t *stack, void *arg)

This pushes the specified argument onto the stack. If this push operation exceeds the size of the stack argument, the stack is not changed and an error is returned.

grpc_error_t grpc_stack_pop(grpc_arg_stack_t *stack, void ** arg)

This removes the top element from the stack and returns it as the argument. If the stack is empty, a null pointer is returned in **arg**.

grpc_error_t grpc_stack_destruct(grpc_arg_stack_t *stack)

This frees all content associated with the specified argument stack.

4.6 Asynchronous GridRPC Control Functions

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

grpc_error_t grpc_probe(grpc_sessionid_t sessionID)

This checks whether the asynchronous GridRPC call has completed.

grpc_error_t grpc_probe_or(grpc_sessionid_t *idArray, size_t length, grpc_sessionid_t *idPtr)

This call checks the array of session IDs for any GridRPC calls that have completed. If any calls have completed, the function return value is **GRPC_NO_ERROR** and exactly one session ID is returned in **idPtr**. If no call has completed, the function return value is also **GRPC_NO_ERROR** but **idPtr** is null. If any of the session IDs in **idArray** are invalid, no operations will occur and an **GRPC_INVALID_SESSION_ID** error will be returned. However, the array of session IDs may contain completed session IDs without causing an error.

Rationale:
Users will typically fill a such an array with session IDs and then check for them to finish one by one. Hence, it will be a common occurrence that such an array may contain completed session IDs. If having a sparse array presents a performance concern, the user has the option of packing the array themselves.
End of Rationale.

grpc_error_t grpc_cancel(grpc_sessionid_t sessionID)

This cancels the specified asynchronous GridRPC call.

grpc_error_t grpc_cancel_all(void)

This cancels all outstanding asynchronous GridRPC calls.

Rationale:
A “cancel array” call was considered but dismissed since it would cause difficult error handling.
End of Rationale.

4.7 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.

Advice to Implementors:
From an implementation standpoint, such information could be conveyed to the OS scheduler to reduce cycles wasted on polling.
End of Advice to Implementors.

grpc_error_t grpc_wait(grpc_sessionid_t sessionID)

This blocks until the specified non-blocking requests to complete.

grpc_error_t grpc_wait_and(grpc_sessionid_t *idArray, size_t length)

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

grpc_error_t grpc_wait_or(grpc_sessionid_t *idArray, size_t length, grpc_sessionid_t *idPtr)

This blocks until *any* of the specified non-blocking requests in a given set has completed.

grpc_error_t grpc_wait_all(void)

This blocks until *all* previously issued non-blocking requests have completed.

grpc_error_t grpc_wait_any(grpc_sessionid_t *idPtr)

This blocks until *any* previously issued non-blocking request has completed.

For **grpc_wait_or()** and **grpc_wait_any()**, exactly one session ID for a completed call is returned in **idPtr**. If more than one call has completed, it is undefined which session ID is returned. That is to say, if more than one call has completed, it is not guaranteed that the session ID returned is for the call that actually completed first.

Error Code Identifier	Notes
GRPC_NOERROR	
GRPC_NOT_INITIALIZED	
GRPC_CONFIGFILE_NOT_EXIST	
GRPC_CONFIGFILE_ERROR	
GRPC_SERVER_NOT_FOUND	
GRPC_FUNCTION_NOT_FOUND	
GRPC_INVALID_FUNCTION_HANDLE	
GRPC_INVALID_SESSION_ID	
GRPC_RPC_REFUSED	
GRPC_COMMUNICATION_FAILED	
GRPC_SESSION_FAILED	
GRPC_STACK_NOT_INITIALIZED	
GRPC_STACK_OVERFLOW	
GRPC_EXCEED_LIMIT	
GRPC_INVALID_LENGTH	
GRPC_OTHER_ERROR_CODE	
GRPC_UNKNOWN_ERROR_CODE	
GRPC_LAST_ERROR_CODE	

Table 2. GridRPC Error Codes

4.8 Error Codes and Error Reporting Functions

When a GridRPC call fails, an error code is returned. Table 2 gives the error code identifiers that can be used with variables of type *grpc_error_t*. These error codes satisfy:

$$0 = \text{GRPC\_NOERROR} < \text{GRPC\_...} < \text{GRPC\_LAST\_ERROR\_CODE}$$

This specifies a useful numerical ordering of the error codes based on the set of integers without specifying a specific implementation.

The ability to check the error code of previously submitted requests is provided. The following error reporting functions provide error codes and human-readable error descriptions. These error descriptions can be more informative about the actual cause of the error.

char *grpc_error_string(grpc_error_t error_code)

This returns the error description string, given a GridRPC error code. If the error code is unrecognized for any reason, the string **GRPC_UNKNOWN_ERROR_CODE** is returned.

grpc_error_t grpc_get_error(grpc_sessionid_t sessionID)

This returns the error code associated with a given non-blocking request.

Rationale:

The GridRPC error codes are intended to be similar to error classes in the MPI standard. That is to say, these are types of errors that are inherent to the GridRPC API and may occur in any GridRPC implementation. Implementation-specific error information may be contained in the associated error description strings. The **GRPC_OTHER_ERROR_CODE** error code may be used for implementation-specific errors.
End of Rationale.

5. Related Work

The concept of Remote Procedure Call (RPC) has been widely used in distributed computing and distributed systems for many years [4]. It provides an elegant and simple abstraction that allows distributed components to

communicate with well-defined semantics. RPC implementations face a number of difficult issues, including the definition of appropriate Application Programming Interfaces (APIs), wire protocols, and Interface Description Languages (IDLs). Corresponding implementation choices lead to trade-offs between flexibility, portability, and performance.

A number of previous works has focused on the development of high performance RPC mechanisms either for single processors or for tightly-coupled homogeneous parallel computers such as shared-memory multiprocessors [7, 3, 13, 2]. A contribution of those works is to achieve high performance by providing RPC mechanisms that map directly to low-level O/S and hardware functionalities (e.g. to move away from implementations that were built on top of existing message passing mechanisms as in [5]). By contrast, GridRPC targets heterogeneous and loosely-coupled systems over wide-are networks, raising a different set of concerns and goals.

This current work grew out of the Advanced Programming Models Research Group [10]. This group surveyed and evaluated many programming models [11, 12], including GridRPC. Some representative GridRPC systems are NetSolve [6], and Ninf [14]. Historically, both projects started about the same time, and in fact both systems facilitate similar sets of features. A number of related experimental systems exist, such as RCS [1] and Punch [15]. Those systems seek to provide ways for Grid users to easily send requests to remote application servers from their desktop. GridRPC seeks to unify those efforts.

This work is also related to the XML-RPC [20] and SOAP [19] efforts. Those systems use HTTP to pass XML fragments that describe input parameters and retrieve output results during RPC calls. In scientific computing, parameters to RPC calls are often large arrays of numerical data (e.g. double precision matrices). The work in [9] made it clear that using XML encoding has several caveats for those types of data (e.g. lack of floating-point precision, cost of encoding/decoding). Nonetheless, recent work [18] has shown that GridRPC could be effectively built upon future Grid software based on Web Services such as OGSA [8].

6. Security

Security issues are not discussed in this document.

7. Author Contact Information

Hidemoto Nakada
Natl. Inst. of Advanced Industrial Science and Technology
hide-nakada@aist.go.jp

Satoshi Matsuoka
Tokyo Institute of Technology
National Institute of Informatics
matsu@is.titech.ac.jp

Keith Seymour
Univ. of Tennessee, Knoxville
seymour@cs.utk.edu

Jack Dongarra
Univ. of Tennessee, Knoxville
dongarra@cs.utk.edu

Craig A. Lee
The Aerospace Corporation, M1-102
2350 E. El Segundo Blvd.
El Segundo, CA 90245
lee@aero.org

Henri Casanova
University of California, San Diego
San Diego Supercomputing Center
casanova@cs.ucsd.edu

Intellectual Property Statement

Full Copyright Notice

References

[1] P. Arbenz, W. Gander, and M. Oettli. The Remote Computation System. *Parallel Computing*, 23:1421–1428, 1997.

[2] I. Aumage, L. Boug, A. Denis, J.-F. Mhaut, G. Mercier, R. Namyst, and L. Prylli. Madeleine II: A Portable and Efficient Communication Library for High-Performance Cluster Computing. In *Proceedings of the IEEE Intl Conference on Cluster Computing (Cluster 2000)*, pages 78–87, 2000.

[3] B. Bershad, T. Anderson, E. Lazowska, and H. Levy. Lightweight Remote Procedure Call. *ACM Transactions on Computer Systems (TOCS)*, 8(1):37–55, 1990.

[4] A. Birrel and G. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[5] L. Boug, J.-F. Mhaut, and R. Namyst. Efficient Communications in Multithreaded Runtime Systems. In *Proceedings of the 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP’99), volume 1568 of Lecture Notes in Computer Science, Springer Verlag*, pages 468–484, 1999.

[6] H. Casanova and J. Dongarra. NetSolve: A Network Server for Solving Computational Science Problems. In *Proceedings of Super Computing ’96*, 1996.

[7] C.-C. Chang, G. Czajkowski, and T. von Eicken. MRPC: A High Performance RPC System for MPMD Parallel Computing. *Software - Practice and Experience*, 29(1):43–66, 1999.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. http://www.globus.org/ogsa, January 2002.
[9] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SC'2000, Dallas, TX*, 2000.
[10] Grid Forum Advanced Programming Models Working Group. Web site. *http://www.eece.unm.edu/~apm*, 2000.
[11] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. http://www.eece.unm.edu/~apm/docs/APM_Primer_0801.pdf, August 2001.
[12] C. Lee and D. Talia. Grid programming models: Current tools, issues and directions. In Berman, Fox, and Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*, pages 555–578. Wiley, 2003.
[13] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC*, Dec. 1993.
[14] H. Nakada, M. Sato, and S. Sekiguchi. Design and Implementations of Ninf: towards a Global Computing Infrastructure. *Future Generation Computing Systems, Metacomputing Issue*, 15(5-6):649–658, 1999.
[15] The Punch project at Purdue. http://punch.ecn.purdue.edu.
[16] K. Seymour et al. An Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *3rd International Workshop on Grid Computing*, volume 2536, pages 274–278. Springer-Verlag, Lecture Notes in Computer Science, November 2002.
[17] K. Seymour et al. GridRPC: A Remote Procedure Call API for Grid Computing. http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf, July 2002.
[18] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services Based Implementations of GridRPC. In *Proc. of HPDC11*, pages 237–245, 2002.
[19] Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP, May 2000. W3C Note.
[20] XML-RPC. http://www.xml-rpc.com.

Contents