GWD-I (Informational)
Advanced Programming Models Research Group
http://www.eece.unm.edu/~apm/docs/APM_GridRPC_0702.pdf

H. Nakada, NIAIST
S. Matsuoka, Tokyo Institute of Technology
K. Seymour, Univ. of Tennessee, Knoxville
J. Dongarra, Univ. of Tennessee, Knoxville
C. Lee, The Aerospace Corp.
H. Casanova, UCSD, SDSC
July 2002

GridRPC: A Remote Procedure Call API for Grid Computing

1. Abstract

This paper discusses preliminary work on standardizing and implementing a remote procedure call (RPC) mechanism for grid computing. The GridRPC API is designed to address one of the factors that has hindered widespread acceptance of grid computing – the lack of a standardized, portable, and simple programming interface. In this paper, we examine two concrete implementations of the GridRPC API based on two different grid computing systems: NetSolve and Ninf. Our initial work on GridRPC shows that client access to existing grid computing systems such as NetSolve and Ninf can be unified via a common API, a task that has proven to be problematic in the past. In addition to these existing grid computing systems, the minimal API defined in this paper provides a basic mechanism for implementing a wide variety of other grid-aware applications and services.

2. Introduction

Although Grid computing is regarded as a viable next-generation computing infrastructure, its widespread adoption is still hindered by several factors, one of which is the question “how do we program on the Grid (in an easy manner)”. Currently, the most popular middleware infrastructure, the Globus toolkit, by and large provides the basic, low-level services, such as security/authentication, job launching, directory service, etc. Although such services are an absolute necessity especially provided as a common platform and abstractions across different machines in the Grid for interoperability purposes (as such it could be said that Globus is a GridOS), there still tends to exist a large gap between the Globus services and the programming-level abstractions we are commonly used to. This is synonymous to the early days of parallel programming, where the programming tools and abstractions available to the programmers were low-level libraries such as (low-level) message passing and/or thread libraries. In a metaphoric sense, programming directly on top of only Globus I/O can be regarded as

performing parallel programming using only the Linux API on a beowulf cluster.

By all means there have been various attempts to provide a programming model and a corresponding system or a language appropriate for the Grid. Many such efforts have been collected and catalogued by the Advanced Programming Models Research Group of the Global Grid Forum [16]. One particular programming model that has proven to be viable is an RPC mechanism tailored for the Grid, or "GridRPC". Although at a very high level view the programing model provided by GridRPC is that of standard RPC plus asynchronous coarse-grained parallel tasking, in practice there are a variety of features that will largely hide the dynamicity, insecurity, and instability of the Grid from the programmers. These are namely:

- Ability to cope with medium to coarse-grained calls, with call durations ranging from $>$ 1 second to $<$ 1 week.
- Various styles of asynchronous, task-parallel programming on the Grid, with thousands of scalable concurrent calls.
- "Dynamic" RPC, e.g., dynamic resource discovery and scheduling.
- Scientific datatypes and IDL, e.g., large matrices and files as arguments, call-by-reference and shared-memory matrix arguments with sections/strides as part of a "Scientific IDL".
- Grid-level dependability and security, e.g., grid security with GSI, automated fault tolerance with checkpoint/rollback and/or retries.
- Simple client-side programming and management, i.e., no client-side IDL management and very little state left on the client.
- Server-side-only management of IDLs, RPC stubs, "gridified" executables, job monitoring, control, etc.
- Very (bandwidth) efficient—does not send entire matrix when strides and array-sections are specified.

As such, GridRPC allows not only enabling individual applications to be distributed, but also can serve as the basis for even higher-level software substrates such as distributed, scientific components on the Grid. Moreover, recent work [21] has shown that GridRPC could be effectively built upon future Grid software based on Web Services such as OGSA [12].

Some representative GridRPC systems are Netsolve [9], and Ninf [18]. Historically, both projects started about the same time, and in fact both systems facilitate similar sets of features as described above. On the other hand, because of differences in the protocols and the APIs as well as their functionalities, interoperability between the two systems has been poor at best. There had been crude attempts at achieving interoperability between the two systems using protocol translation via proxy-like adapters [18], but for various technical reasons full support of mutual features proved to be difficult.

This experience motivated the need for a more unified effort by both parties to understand the requirements of the GridRPC API, protocols, and features, and come to a common ground for potential standardization. In fact, as the Grid became widespread, the need for a unified standard GridRPC became quite apparent, in the same manner as MPI standardization, based on past experiences with different message passing systems, catapulted the adoption of portable parallel programming on large-scale MPPs and clusters.

This paper reports on the current status of GridRPC standardization. Based on the lessons and experiences learned from the MPI standardization process as well as deployment of respective systems, both groups determined several design criteria as follows:

1. A small team of people experienced in GridRPC design and deployment would collaboratively design the API, taking into account the current RPC designs from Netsolve and Ninf as well as existing RPC standards such as CORBA.
2. The initial goal is to standardize the API so that programmers can assume portability of their source across the platforms. The protocol standardization is more difficult and will be dealt with eventually (this is the same situation with MPI, and CORBA until IIOP was standardized.)
3. Define a minimal set of features first, then investigate if higher-level features could be built on top of the minimal features and their API.
4. Have several reference implementations, if possible, based on existing Netsolve/Ninf code, or even a new code base.

The rest of the paper will describe the fundamental features of the GridRPC model, the proposed standard API, and the details of two reference implementations.

3. The GridRPC Model and API

In this section, we informally describe the GridRPC model and the functions that comprise the API. Appendix A contains a detailed listing of the function prototypes.

3.1 Function Handles and Session IDs

Two fundamental objects in the GridRPC model are *function handles* and the *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.

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

3.3 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).

3.4 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*.

- `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.
- `grpc_call_argstack` makes a blocking call using the argument stack.
- `grpc_call_argstack_async` makes a non-blocking call using the argument stack.

3.5 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_cancel` cancels the specified asynchronous GridRPC call.

3.6 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.

3.7 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_perror` prints the error string associated with the last GridRPC call.
- `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_last_error` returns the error code for the last invoked GridRPC call.

3.8 Argument Stack Functions

When describing the GridRPC call functions, we mentioned that there is an alternate calling style that uses an *argument stack*. With the following functions it is possible to construct the arguments to a function call at run-time. 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);`.

- `newArgStack` creates a new argument stack.
- `pushArg` pushes the specified argument onto the stack.
- `popArg` removes the top element from the stack.
- `destructArgStack` frees the memory associated with the specified argument stack.

4. Implementations

Since the GridRPC interface does not dictate the implementation details of the servers which execute the procedure call, there may be multiple different implementations of the GridRPC API, each having the ability to communicate with one or more Grid computing systems. In fact, having multiple implementations is desirable because it allows GridRPC to fulfill its goal of unifying different existing systems. In this section, we describe two implementations of the GridRPC API, one implemented on top of NetSolve and the other on top of Ninf.

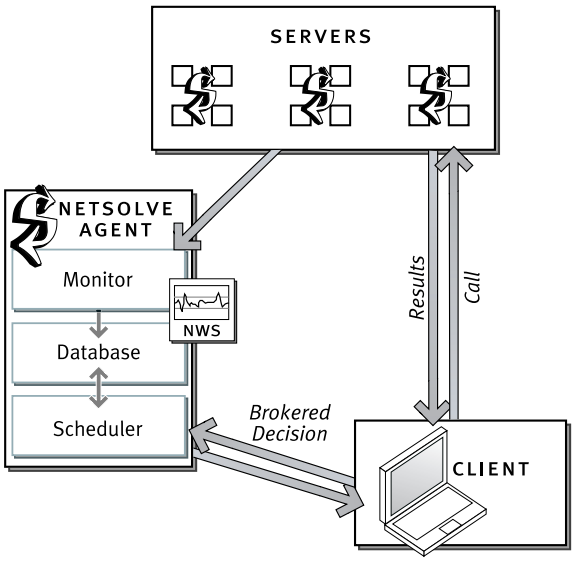

Figure 1. Overview of NetSolve

4.1 GridRPC over NetSolve

NetSolve [4] is a client-server system which provides remote access to hardware and software resources through a variety of client interfaces, such as C, Fortran, and Matlab. Since NetSolve's mode of operation is in terms of RPC-style function calls, it provides much of the infrastructure needed to implement GridRPC.

4.1.1 Overview of NetSolve

A NetSolve system consists of three entities, as illustrated in Figure 1.

- The *Client*, which needs to execute some function remotely. In addition to C and Fortran programs, the NetSolve client may be an interactive problem solving environment, such as Matlab or Mathematica.
- The *Server* executes functions on behalf of the clients. The server hardware can range in complexity from a uniprocessor to a MPP system and similarly the functions executed by the server can be arbitrarily complex. Server administrators can straightforwardly add their own software without affecting the rest of the NetSolve system.
- The *Agent* is the focal point of the NetSolve system. It maintains a list of all available servers and performs resource selection for all client requests as well as ensuring load balancing of the servers.

In practice, from the user's perspective the mechanisms employed by NetSolve make the remote call fairly transparent. However, behind the scenes, a typical call to NetSolve involves several steps, as follows:

1. The client queries the agent for an appropriate server that can execute the desired function.

2. The agent returns a list of available servers, ranked in order of suitability.

3. The client attempts to contact a server from the list, starting with the first and moving down through the list. The client then sends the input data to the server.

4. Finally the server executes the function on behalf of the client and returns the results.

4.1.2 Using NetSolve to Implement GridRPC

Currently we have a full implementation of the GridRPC API running on top of the NetSolve system. An important factor in enabling the implementation of GridRPC in NetSolve is the strong similarity of their APIs. For example, `grpc_call()` and `grpc_call_async()` map directly into the `netsolve()` and `netsolve_nb()` calls. `grpc_probe()` and `grpc_cancel()` map into the `netslpr()` and `netslkill()` calls. Some of the other GridRPC functions that do not map directly to the NetSolve API can be implemented in terms of those that do. For example, `grpc_wait_and`, `grpc_wait_or`, `grpc_wait_any`, and `grpc_wait_all`, are all implemented using the elementary `grpc_wait` function. Some GridRPC functions cannot be expressed in terms of another existing function, so we implemented them from scratch. The function handle creation and destruction functions fall into that category since the function handle concept does not exist in NetSolve. Also, the argument stack calling sequence required some slight modification to the NetSolve client because it previously only supported the variable argument list calling sequence.

Besides the advantageous similarity in these APIs, NetSolve has several properties that make it an attractive choice for implementing GridRPC: fault-tolerance, load-balancing, and security.

NetSolve handles fault detection and recovery in a way that is transparent to the user. The agent is constantly monitoring the status of all the servers so that in case of a problem, the agent can choose a new server to handle the problem. The client software submits the problem to the new server, but the user is unaware of the resubmission, similar to the way that the user of a TCP socket is unaware of the retransmission of packets. To facilitate detection of server failures and network problems, NetSolve has integrated the

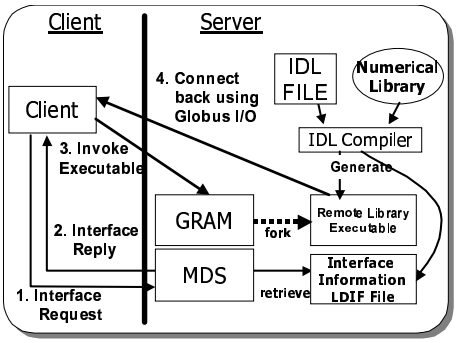

Figure 2. Overview of Ninf-G

Network Weather Service [25] and the Heart Beat Monitor [23] from Globus.

NetSolve strives to schedule the use of the computational resources in the most efficient manner possible. To that end, NetSolve employs a load-balancing strategy that takes into account several system parameters, such as network bandwidth and latency, server workload and performance, and complexity of the function to be executed. The NetSolve agent uses this load-balancing algorithm to select the most suitable server to execute a given request, thereby providing the user with the best response time as well as maintaining balanced usage of all the hardware resources.

Starting with version 1.4, NetSolve has support for basic Kerberos authentication. Kerberos is a network authentication protocol "designed to provide strong authentication for client/server applications by using secret-key cryptography". Using Kerberos, the NetSolve client must prove its identity to the server before being allowed to execute a task on that server.

4.2 GridRPC over Ninf

4.2.1 Overview of Ninf-G

Ninf-G is a re-implementation of the Ninf system [18] on top of the Globus Toolkit [11]. The Globus toolkit provides a reference implementation of standard (or subject to proposed standardization) protocols and APIs for Grid computing. Globus serves as a solid and common platform for implementing higher-level middleware and programming tools, etc., ensuring interoperability amongst such high-level components, one of which is Ninf-G.

Figure 2 shows an overview of the Ninf-G system in this regard.

Ninf-G is designed focusing on simplicity. In contrast with NetSolve, Ninf-G does not provide fault detection, recovery or load-balancing by itself. Instead, Ninf-G assumes

that backend queuing system, such as Condor[20], takes responsibility for these functionality.

Ninf-G fully deploys Globus Security Infrastracture. It means that not only all the components are protected propery, but also they can utilize other Globus components, such as GridFTP servers, seamlessly and securely.

Client API. Largely speaking, Ninf-G has two categories of API. One is the *GridRPC* API which is discussed in this paper, and another is the *Ninf API* which is provided only for compatibility with the old Ninf system. Ninf-G also provides various other tools to "gridify" libraries and applications, such as a compile driver which automates the compilation and linkage of Ninf-G client programs.

Server side IDL. In order to "gridify" a library, the Ninf library provider describes the interface of the library function using the Ninf IDL to publish his library function, which are only manifested and handled at the server side. Besides supporting access specifiers such as `IN` and `OUT` denoting whether an argument is read or written, the Ninf IDL supports datatypes mainly tailored for serving numerical applications. For example, the basic datatypes include scalars and their multi-dimensional arrays. There are also special provisions such as support for expressions involving input arguments to compute array sizes, designation of temporary array arguments that need to be allocated on the server side but not transferred, etc. This allows direct "gridifying" of existing libraries that assume array arguments to be passed by call-by-reference (thus requiring shared-memory support across nodes via software), and supplementing the information lacking in the C and Fortran type-systems regarding array sizes, array stride usage, array sections, etc.

Ninf-G and the Globus toolkit. Ninf-G employs the following components from the Globus toolkit.

- *GRAM* (Globus Resource Allocation Manager) is a "secure inetd" which authenticates clients using GSI-based certificates, maps to the local user account, and invokes executable files.
- *MDS* (Monitoring and Discovering Service) is a directory service to provide resource information within the Grid. Ninf-G uses the MDS to publish interface information about the GridRPC components.
- *Globus-I/O* enables secure communication using GSI, providing blocking and non-blocking I/O that is integrated with Globus threads. In Ninf-G, the client and remote executable communicate with each other using Globus-I/O.

4.2.2 Using Ninf-G to Implement GridRPC

As in NetSolve, the Ninf-G design allows direct support for the GridRPC model and API. The steps in making an actual Ninf-G GridRPC call can be broken down into those shown in Figure 2.

1. *Retrieval of interface information and executable pathname.* The client retrieves this information registered in the MDS using the library signature as a key. The retrieved info is cached in the client program to reduce the MDS retrieval overhead.
2. *MDS sends back the requested information.*
3. *Invoking remote executable.* The client invokes the remote executable via the Globus GRAM, specifying the remote executable path obtained from the MDS and a port address that accepts the callback from the remote executable. Here, the accepting port authenticates its peer using Globus-I/O, preventing malicious third party attacks as only the party that owns the proper Globus proxy certificates derived from the client user certificate can connect to the port.
4. *Remote executable callbacks to the client.* The remote executable obtains the client address and the port from the argument list and connects back to the client using Globus-I/O for subsequent parameter transfer, etc. Subsequent remote executable communication with the client will use this port.

5. Related Work

The concept of Remote Procedure Call (RPC) has been widely used in distributed computing and distributed systems for many years [7]. 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 [10, 6, 17, 5]. 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 [8]). By contrast, our work on GridRPC targets heterogeneous and loosely-coupled systems over wide-are networks, raising a different set of concerns and goals.

A number of technologies provide ways for applications to be structures as sets of *distributed objects*, such as CORBA and Java RMI, where those objects communicate via remote method invocations. Therefore, those systems support RPC programming. However, their goal is much broader, which comes at the expense of software simplicity and light footprint, which are both among our goals. In previous work, we have conducted quantitative and qualitative comparisons of CORBA technology with our NetSolve and Ninf systems, in the context of RPC programming for scientific computing [24]. We found several compelling reasons (namely IDL complexity, IDL expressiveness, protocol performance, software footprints) not to re-use distributed object technology, but rather to focus on a simple, lightweight implementation of RPC functionality that meets the needs of scientific computing.

A number of experimental systems are related to our work on NetSolve and Ninf, such as RCS [3] and Punch [19]. Those systems seek to provide ways for Grid users to easily send requests to remote application servers from their desktop. Our work on GridRPC seeks to unify those efforts. This paper takes the first step by proposing a recommendation for a standard GridRPC API. Another key component of an RPC system is its IDL. NetSolve and Ninf both have two different IDLs with different tradeoffs between complexity and expressiveness. We are currently working on an IDL definition for GridRPC. Unlike CORBA, we do not require that client software be upgraded (re-compiled) with new RPC stubs when servers offer new services. This requires that the IDL stubs (or at list the section that is used on the client side for argument marshaling) be downloaded and executed at runtime. NetSolve and Ninf provide this capability with simple runtime interpretation of the IDL language. Another approach is for IDL stubs to contain code that can be dynamically linked and executed, as it is done in Jini. For reasons of cross-language portability, we believe that the GridRPC IDL should follow the Ninf/NetSolve model. We are currently investigating XML schemas for the GridRPC IDL.

This work is also related to the XML-RPC [26] and SOAP [22] 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 [14] 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). A solution is to use a hybrid protocol that may use an XML skeleton to describe data being sent, but that would send binary data as "attachments". Based on the NetSolve and the Ninf protocols, we are currently defining a GridRPC wire protocol.

Finally, our work on GridRPC fits in the framework of the Global Grid Forum Research Group on Programming Models [13, 15]. That venue allows us to communicate our proposals and findings to the Grid community.

6. Discussion and Conclusions

We have presented a preliminary work in defining a model and API for a grid-aware RPC mechanism. Besides enabling individual applications to be distributed and allowing the different parts to interact, remote procedure invocation is a fundamental capability that will enable many other capabilities to be built. Such capabilties include network-enabled services that are persistent and discoverable in the environment, and component architectures where predefined or application-specific components must interact through well-known ports or interfaces. The inherent nature of invoking a remote procedure across a network connection (rather than on a stack) means that only coarse grain calls will be appropriate and that computation/communication ratios will be a driving factor. This reflects the fundamental fact that grid environments present a heterogeneous communication hierarchy across machines and networks.

In all software systems, there is a fundamental choice between performance and flexibility. The choice was made here to preserve performance rather than adopt a very flexible but heavyweight protocol based on XML document transfer. This does not preclude the use of XML internally but it also does not require its use by exposing it through the API. While this may currently limit the ease of adaptation for GridRPC codes, GridRPC is now very bandwidth efficient which can be a key issue for large-scale, high-performance applications. We note that as XML evolves, it may eventually allow binary fields of arbitrary length, at which point, its use may become more attractive.

While the model and API presented here is a first-step towards a general GridRPC capability, there are certainly a number of outstanding issues regarding wide-spread deployment and use. The first is simply *discovery*. Currently a remote procedure is discovered by explicitly asking a well-known server for a well-known function through a name string lookup. Establishing this function-to-server mapping is all that the user cares about and, hence, the GridRPC model does not define how discovery is done. For a wide variety of applications, domains, and institutions, a straightforward discovery mechaism such as the NetSolve Agent will be completely sufficient. Other applications, however, may need to look for appropriate servers over a wider, open-ended grid environment. In this case, discovery via the Globus MDS may be more suitable. Applications may also want to request functions or services by *type* rather than

name. In this case, function signature meta-data schemas will have to be defined to facilitate such discovery. Hence, while the GridRPC API should not define how discovery is done, there may be a need for an application to express general discovery constraints.

Scheduling is also another issue. Currently individual RPCs are processed independently. The actual scheduling of the remote invocation is unilaterally determined by the daemon honoring the RPC request. Clients, however, may have scheduling constraints to meet their processing requirements. If a remote call entails submitting a batch job, the client may at least want to know what the queue length is, or have some notion of the expected time of completion. Clients may also need to *co-schedule* multiple RPCs. A client may not want to schedule two (or more) RPCs unless they can be scheduled at the same time, or at least within some constraint. While co-scheduling is a fundamental capability, the decreased probability of being able to succesfully co-schedule (especially on difficult-to-acquire resources) will limit its use to those cases that absolutely require it.

At this early stage of development and use, applications will only use GridRPC in shallow call trees, e.g., making one call to a service provided at one remote location. As such a capability becomes more stable and available, however, it is conceiveable that applications will be built with arbitrary call depths. While *fault-tolerance* and *security* are important for shallow cases, call trees of arbitary depth will require some notion of *transitive* or *composible* fault-tolerance and security.

Currently fault-tolerance is accomplished by checkpoints, rollbacks and retries. In any larger, distributed environment, an event service may be useful to manage cancellations and rejections along a call tree and other such aspects.

Security will require a transitive *delegation* of trust as described in [1] and [2]. We note that cancellation of a secure RPC could require the revocation of delegated trust. This is currently not considered in these documents. Signing and checking certificates on an RPC represents an overhead that must be balanced against the amount of work represented by the RPC. Security overheads could be managed by establishing secure, trusted domains. RPCs within a domain could dispense with certificates; RPCs that cross domains would have to use them. Trusted domains could be used to limit per-RPC security overheads in favor the one-time cost of establishing the domain.

While these larger issues may be on the horizon, they should not be allowed to overshadow the importance of the development and use of a practical GridRPC capability. Such a capability will produce a body of experience that will sort out the priorities for future work.

7. Security

A brief discussion of general security issues appears in Section 6.

8. Author Contact Information

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

Satoshi Matsuoka
Tokyo Institute of Technology
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

References

[1] Internet X.509 Public Key Infrastructure Proxy Certificate Profile. `http://www.ietf.org/internet-drafts/draft-ietf-pkix-proxy-01.txt`.

[2] TLS Delegation Protocol. `http://www.ietf.org/internet-drafts/draft-ietf-tls-delegation-01.txt`.

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

[4] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4. Computer Science Dept. Technical Report CS-01-467, University of Tennessee, Knoxville, TN, July 2001.

[5] 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.

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

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

[8] 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.

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

[10] 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.

[11] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 1997.

[12] 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.

[13] Global Grid Forum Research Group on Programming Models. `http://www.gridforum.org/7_APM/APS.htm`.

[14] 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.

[15] C. Lee. Grid RPC, Events, and Messaging, Sept. 2001. Informational Grid Working Draft (GWD-I) available at `http://www.eece.unm.edu/~apm/WhitePapers/APM_Grid_RPC_0901.pdf`.

[16] C. Lee, S. Matsuoka, D. Talia, A. Sussman, M. Mueller, G. Allen, and J. Saltz. A Grid Programming Primer. http://www.gridforum.org/7_APM/APS.htm, submitted to the Global Grid Forum, August 2001.

[17] J. Liedtke. Improving IPC by Kernel Design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP), Asheville, NC*, Dec. 1993.

[18] 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.

[19] The Punch project at Purdue. `http://punch.ecn.purdue.edu/`.

[20] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proc. of HPDC-7*, 1998.

[21] S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services Based Implementations of GridRPC. In *Proc. of HPDC11 (to appear)*, 2002.

[22] Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP/`, May 2000. W3C Note.

[23] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *7th IEEE Symp on High Performance Distributed Computing*, pages 268–278, 1998.

[24] T. Suzumura, T. Nakagawa, S. Matsuoka, H. Nakada, and S. Sekiguchi. Are Global Computing Systems Useful? Comparison of Client-Server Global Computing Systems Ninf, NetSolve Versus CORBA. In *Proceedins of the 14th Parallel and Distributed Processing Symposium, IPDPS'00*, pages 547–559, May 2000.

[25] R. Wolski. Dynamically Forecasting Network Performance to Support Dynamic Scheduling Using the Network Weather Service. In *6th High-Performance Distributed Computing Conference*, August 1997.

[26] XML-RPC. `http://www.xml-rpc.com/`.

A GridRPC API Specification

A.1 Initializing and Finalizing Functions

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

A.2 Remote Function Handle Management Functions

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

A.3 GridRPC Call Functions

```
int grpc_call(grpc_function_handle_t *handle, ...);
int grpc_call_async(grpc_function_handle_t *handle, ...);
int grpc_call_argstack(grpc_function_handle_t *handle, ArgStack *args);
int grpc_call_argstack_async(grpc_function_handle_t *handle, ArgStack *args);
```

A.4 Asynchronous GridRPC Control Functions

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

A.5 Asynchronous GridRPC Wait Functions

```
int grpc_wait(int sessionID);
int grpc_wait_and(int * idArray, int length);
int grpc_wait_or(int * idArray, int length, int * idPtr);
int grpc_wait_all();
int grpc_wait_any(int * idPtr);
```

A.6 Error Reporting Functions

```
void   grpc_perror(char * str);
char * grpc_error_string(int error_code);
int    grpc_get_error(int sessionID);
int    grpc_get_last_error();
```

A.7 Argument Stack Functions

```
ArgStack *newArgStack(int maxsize);
int       pushArg(ArgStack *stack, void *arg);
void     *popArg(ArgStack *stack);
int       destructArgStack(ArgStack *stack);
```