A WSRF-Compliant Debugger for Grid Applications

Donny Kurniawan and David Abramson
Monash e-Science and Grid Engineering Lab
Faculty of Information Technology, Monash University
{donny.kurniawan, david.abramson}@infotech.monash.edu.au

Abstract

Grid computing allows the utilization of vast computational resources for solving complex scientific and engineering problems. However, development tools for Grid applications are not as mature as their traditional counterparts, especially in the area of debugging and testing. Debugging Grid applications typically requires a programmer to address non-trivial issues such as heterogeneity, job scheduling, hierarchical resources, and security. This paper presents the design and implementation of a Grid service debug architecture that is compliant with the Web Service Resource Framework standard. The debugger provides a library with a set of well-defined debug APIs.

1. Introduction

Grid computing facilitates the aggregation of geographically distributed resources such as data servers, compute clusters, and scientific instruments to act as a single large system. This virtualisation allows multiple organizations to cooperate on solving large-scale problems which cannot be addressed by a single organization [1]. Grid middleware provides a set of services for managing and controlling the resources and the infrastructure. The services include security, resource management, data management, and information services. This middleware defines protocols and standards for users to manage program execution, to access data, to monitor resources, and to query information [14].

Applications need to be specifically written to take full advantage of the remote resources. However, currently Grid programmers still rely on traditional tools and techniques that are designed for local development. For example, applications are typically written on a programmer's desktop and are transferred to remote resources to be run. Debugging these applications requires the programmer to log in to the remote nodes and to run a local debugger [2]. These development techniques are time consuming and error prone and we believe will hinder the adoption of the Grid.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

In the case of Grid application debugging, the distributed and heterogeneous nature of Grids presents a challenge specifically in the areas of heterogeneity, job scheduling, resource hierarchy, and security.

Heterogeneity: A Grid testbed may consist of resources with different computer architectures, operating systems, and debuggers. Some of these differences are addressed by current Grid middleware such as Globus [10], for example, by providing uniform ways of invoking applications and transferring data files. However, these do not support debugging, and thus the underlying heterogeneity is still exposed to the programmer.

Job scheduling: In a typical Grid environment program execution is managed by a local queue manager, which schedules jobs according to criteria such as resource availability and processor load. This batch processing scenario makes it difficult to debug an application interactively because the programmer cannot easily determine when the job actually starts execution. As a result, programmers must resort to ad-hoc techniques such as polling the scheduler regularly to check whether the application has started, and then they must manually attach a debugger to the process. In a Grid, this process might need to be repeated across multiple resources, possibly using different local schedulers, making the technique cumbersome and error prone.

Hierarchical resources: Many Grid testbeds are actually built from distributed clusters, consisting of a single front-end machine and multiple back-end processor nodes. These back-end nodes are usually only accessible from the front-end, which may in turn be behind a gateway server or a firewall. Thus, debugging an application running on the execution nodes of a cluster may require access to a hierarchy of intermediate computers. However, depending on the Grid security policy in place, a programmer may not have direct access to all of these machines, again complicating debugging.

Security: Grid level debugging must be secure. For example, a debugger run by one user must be restricted from attaching to and controlling another user's processes. Clearly a debugger must conform to the security framework implemented in the Grid middleware. Thus, if the job invocation mechanism uses X.509 certificates, such as used by Globus, then the debugger must also operate in this framework.

We believe that many of these issues can be solved by building the debug architecture into the Grid fabric. This research paper focuses on the specification and design of a standard set of application programming interfaces (APIs) suitable for debugging and testing computational Grid applications. These services can then be used within a Grid level debugger, or other high level software tools that require these functions. The architecture is modular and independent of the particular Grid middleware and back-end debug servers that are used.

The rest of the paper is organized as follows. Section 2 presents the design and architecture of the debugger. The implementation details are described in section 3. In section 4, we outline related work in the area of Grid debugging tools. The discussion on how our approach meets the challenges is given in section 5. Section 6 presents the conclusion.

2. Design and architecture

The debugger is not designed as a stand-alone monolithic program but rather as component-based software which includes a plug-in or an extension to Grid middleware. This design simplifies adding a debugging service to existing middleware. The debugger is composed of four components: the client, the middleware compatibility layer, the debug library, and the debug back-end. The components and the overall architecture are shown in figure 1.

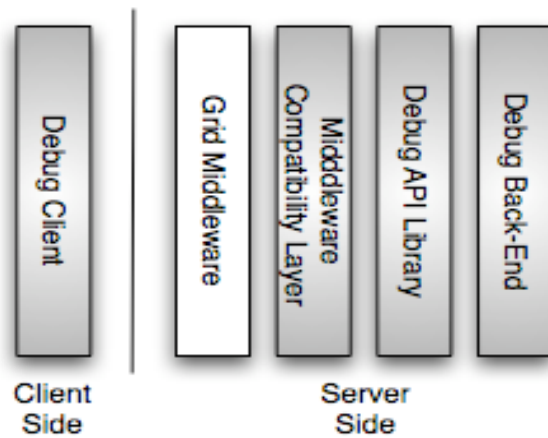

Figure 1. Debugger architecture

Debug client. A developer utilizes the debugging service by using a debug client that is written according to client specification of the Grid middleware. It takes advantage of various services offered by the middleware such as security and notification services. Various APIs in the debug library are utilized by the client, which can then be implemented as a simple command-line interface (CLI) program similar to a traditional CLI debugger or as a plug-in of an IDE such as Eclipse or NetBeans.

Middleware compatibility layer. The compatibility layer is a Grid middleware plug-in that wraps the debug library as a Grid debugging service. The layer is written according to the extension mechanism of the middleware. This component acts as a translation layer between middleware-specific and generic debug interface.

Debug library. The main component of the system is a debug library with a set of well-defined debug APIs based on the High Performance Debugging Forum (HPDF) standard [9]. HPDF was chosen because it is the result of significant research on an appropriate command set for a parallel debugger, and it serves as a sound base for a debug library interface. The API defines a generic debug model and a collection of methods and objects for Grid application debugging. The debug library is middleware-

independent and is linked to the compatibility layer.

HPDF is a collaborative effort in the area of parallel and distributed systems with a goal to define standards for debugging tools for high-performance computers. HPDF has defined a standard for command-line interface parallel debuggers. This standard forms the model of our debug API. The API specifies objects and methods that correspond to commands in the HPDF standard. Not all commands are translated into methods since some of them are only peculiar to CLI tools. The API includes, for example, methods to define process sets, to create/delete breakpoints, and to control the execution of programs. The API is discussed in more detail in section 2.1.

Debug back-end. The debug back-end provides functionalities as defined by the API. It is the component that performs the actual low-level debug operations, for example ptrace system calls, on the debugged application. A traditional debugger such as GDB can be utilized as the back-end debugger by implementing an interface to the debug library.

2.1. Debug API

The debug API specifies a debug model, methods, and data structures using an object-oriented paradigm. Classes are defined to represent entities such as processes, breakpoints, and events. Although the definitions are language-independent, we adopt Java programming language to implement the debug library. The design of the API is based on the HPDF standard. HPDF commands, for example, focus, defset, load, and step are implemented as methods (see [15] pages 20 and 21 for definitions of these). However, there are several key differences between the API and the standard which are discussed below.

CLI-specific features. The HPDF standard also defines CLI-specific features such as command history and debugger state variables. These features relate to the user interface aspect of CLI tools and they are not pertinent to API-based debuggers. The debug API does not keep track of command history, however, such functions can be implemented in the debug client for user convenience. A class, DebugConfig, is provided to store configuration parameters. Various DebugConfig methods can be called to change the behaviour of the API.

Output and events. In response to user input, a debugger typically issues a variety of messages through an output stream to a terminal or to a GUI window. Instead of a stream, the debug API employs an event-based mechanism with a set of methods and a queue. The debug client checks the event queue at regular intervals to retrieve debug messages and output. Alternatively, if the Grid middleware supports a notification service, the client can utilize it to be notified of any messages from the debug library.

Processes and threads. HPDF recognizes three models of parallelism: processes-only, threads-only, and multilevel (multi-process and multi-thread). However, the current implementation of the debug library only supports multi-process debugging. Various methods, for example, focus, defSet, and undefSet are provided that allow programmers to debug multiple processes easily.

Remote debugging. The debug API is augmented with a remote debugging feature. Although not specified in the HPDF standard, the feature gives more flexibility in Grid application debugging. The extension allows a remote debugger such as GDB/GDBServer to be utilized as the back-end debug engine.

Debug sessions. The debug API is accessed through a DebugSession object which represents a single debugging session for a user (figure 2). It comprises objects for event notification (DebugEventManager), for storing configuration parameters (DebugConfig), for remote debugging (RemoteDebugManager), and for accessing the underlying back-end debugger (IDebugger). The API does not include any security-related functionality such as user identification and authorization. Instead, it relies on the security service provided by the Grid middleware. Table 1 lists the currently implemented interfaces.

General Debugger Interface
DebugVariable[] set()
DebugVariable set(String var)
void set(String name, String val)
void unset(String var)
void unsetAll()

```
Process Sets
void focus(String name)
void defSet(String name, int[] procs)
void undefSet(String name)
void undefSetAll()
DebugProcess[] viewSet(String name)

Debugger Initialization/Termination
void load(String prg)
void load(String prg, int numProcs)
void run(String[] args)
void run()
void detach()
void exit()

Program Information
DebugStackFrame[] where()

Data Display and Manipulation
String print(String expr)

Execution Control
void step()
void stepSet(String set)
void step(int count)
void stepSet(String set, int count)
void stepOver()
void stepOverSet(String set)
void stepOver(int count)
void stepOverSet(String set, int
count)
void stepFinish()
void stepFinishSet(String set)
void halt()
void haltSet(String set)
void cont()
void contSet(String set)

Actionpoints
void breakpoint(String loc)
void breakpointSet(String set, String
loc)
void breakpoint(String loc, int
count)
void breakpointSet(String set, String
loc, int count)
void breakpoint(String loc, String
cond)
void breakpointSet(String set, String
loc, String cond)
void watchpoint(String var)
void watchpointSet(String set, String
var)
DebugActionpoint[] actions()
DebugActionpoint[] actions(int[] ids)
DebugActionpoint[] actions(String
type)
void delete(int[] ids)
void delete(String type)
void disable(int[] ids)
void disable(String type)
void enable(int[] ids)
void enable(String type)

Miscellaneous
void remote(String host, int port)
void remoteManagerServe(int infPort,
int supPort)
void remoteManagerDestroy()
boolean
remoteManagerIsInferiorConnected()
boolean
remoteManagerIsSuperiorConnected()
DebugEventQueue getEventQueue()
```

Table 1. Debug API

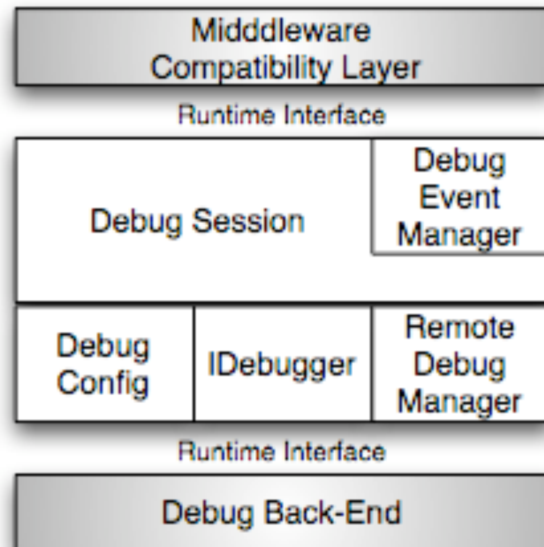

Figure 2. Debug library components

3. Implementation of the debugging service

We have implemented a Grid debugging service using two technologies: Globus Toolkit 4 as the Grid middleware layer and GDB [4] as the back-end debug engine. Sections 3.1 and 3.2 give more descriptions about GT4 and GDB. Section 3.3 presents a detailed explanation and outlines a sequence of events that happen in a debugging session. Section 3.4 discusses two clients of the debugging service.

3.1. Globus Toolkit 4 and WSRF

Globus Toolkit has been developed since the late 1990s to support the development of service-oriented distributed applications and infrastructures [10]. It enables easy federation of distributed resources such as data storage, compute clusters, networks, and remote sensors. The Globus Toolkit is currently the predominant middleware deployed on Grid resources.

The latest release of the toolkit, Globus Toolkit 4 (GT4), employs extensive use of Web Services to define its interfaces and component structures. Web Services are of special interest since they are implementation and platform independent, and their role of interconnecting various systems is similar to the role of Grid computing [11]. However, traditional Web Services are stateless which prevents them from retaining data between invocations. Various workarounds exist such as browser cookies and session identifications to enable stateful Web Services. Nevertheless, these workarounds are non-standard and may not allow communication between services.

WSRF (Web Service Resource Framework) is a set of proposed specifications that provides a standard-defined way to create stateful Web Services [12]. These OASIS-published specifications allow Web Services to retain their states while communicating with each other or with other resources. Globus Toolkit 4 has been designed and implemented around WSRF. Its services are WSRF-compliant and provide API with C and Java bindings.

On GT4 resources, the actual execution of Grid applications is handled by WS-GRAM. WS-GRAM (Web Services Grid Resource Allocation and Management) is the execution manager of Globus Toolkit. It is a set of WSRF-compliant Web Services that allows users to submit, to monitor, and to cancel jobs on Grid resources. WS-GRAM itself is not a job scheduler. It utilizes the standard fork() system call or a local job scheduler, for example, PBS or LSF on the resources, but importantly, these are virtualised and are not visible to the client processes.

3.2. GDB and GDBServer

The GNU Debugger (GDB) [4][13] is the de facto source-level debugger used on many computer architectures. It supports programming languages such as C, C++, and Fortran. GDB can act as a software-controllable back-end debugger through the use of GDB/MI which is a machine oriented text interface to GDB. The interface provides means for a high-level debugger to be built on top of GDB.

An advantage of using GDB as the back-end debug engine is the remote debugging feature which is handled by an auxiliary program called GDBServer. It performs ptrace operations on a debugged program on remote machines. Figure 3 shows the interaction between GDB and GDBServer and how they work:

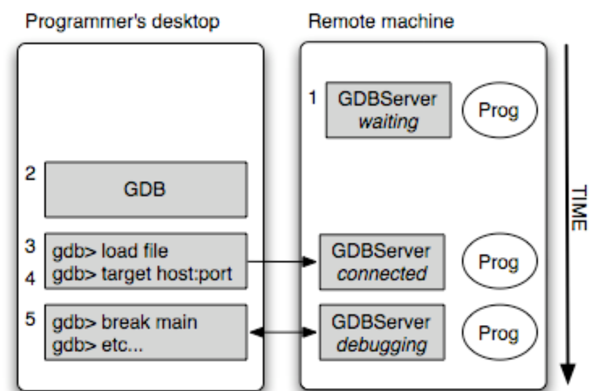

Figure 3. GDB and GDBServer interaction

1. A user logs in to a remote machine and invokes GDBServer with a program to be debugged. GDBServer then waits for a connection from GDB.
2. GDB is started on a programmer's desktop.
3. The debugging symbol file of the program is loaded in GDB.
4. The user instructs GDB to connect to the remote target using a TCP/IP connection. The user specifies the host name and the port number.
5. The user debugs the remote program as usual with GDB as if the debugged program runs on the programmer's desktop.

Remote debugging using GDBServer has some advantages which are listed below:

- The size of GDBServer is very small and it does not overload remote nodes.
- The debugging symbol resolution is performed at the GDB end thus a program on a remote machine does not need to be compiled with symbols included. However, the user must have another copy of the program with the debugging symbols on the local machine.
- GDB and GDBServer support debugging of cross-compiled applications. This allows the user to debug a program on a remote machine which has a different computer architecture from the user's desktop.

Nevertheless, in the Grid environment, GDBServer is not suitable for remote debugging since it does not address issues such as security, job scheduling and hierarchical resources. To solve this problem, we propose implementing a callback notification in GDBServer. We reversed the waiting-connecting mechanism in GDB and GDBServer. In this mechanism, it is GDB that waits for a callback connection from GDBServer. This technique alleviates the need to constantly poll the scheduler to check whether an application has been started. In a testbed with a hierarchy of resources, a programmer does not need to have access to intermediate machines, provided that the back-end node where GDBServer is running can access the programmer's desktop. This technique also adds another layer of security since the debugging activity is initiated by the application rather than by the programmer thus a user cannot debug another user's processes. A sample debugging session to illustrate this is given in section 3.3.

3.3. Debugging a Grid application

An example of debugging a Grid application is shown in figure 4. The description of the sequence of events that happen in the debugging session:

1. A Grid user submits a job to WS-GRAM. The submitted job includes GDBServer and the application. Host address and port number details for GDBServer are also given as arguments. The user must have a valid Globus certificate to access WS-GRAM.
2. WS-GRAM passes the job to a job scheduler. Depending on the scheduling criteria, the job may not be executed immediately.
3. Independently, the user invokes a debug client that interacts with the Globus compatibility layer called WS-DBG. WS-DBG acts as a wrapper for the debug library and exposes the library as a WSRF-compliant debugging service. The library then starts GDB (arrow 6) that waits for a connection from GDBServer. The same X.509 certificate that is used to access WS-GRAM is also required to access WS-DBG.
4. At a scheduled time, the job scheduler executes the job. GDBServer is started to debug the application. Details of the host address and port number are passed to GDBServer.
5. GDBServer contacts GDB using a TCP/IP connection and establishes a two-way communication (arrow 7).

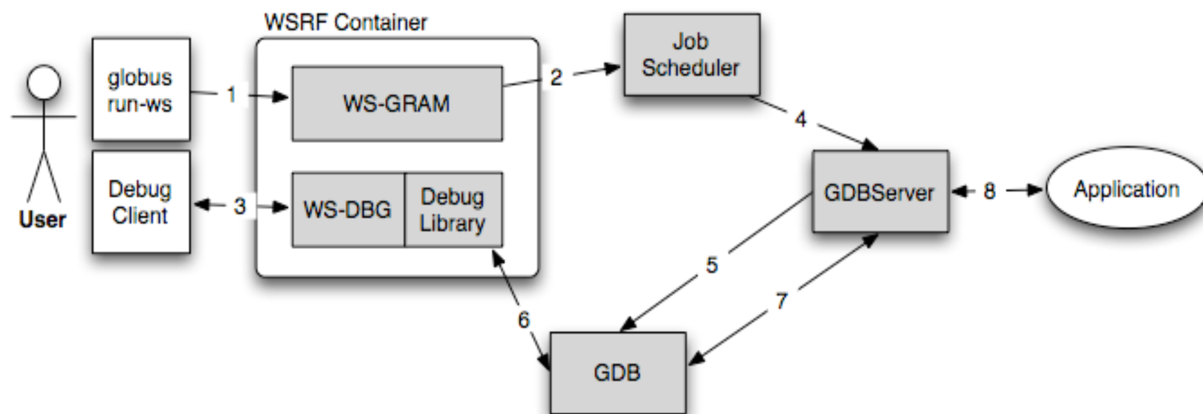

Figure 4. Debugging a Grid application

6. The debug library communicates with GDB. API methods invoked by the user are translated as GDB commands using the GDB/MI interface. GDB instructs GDBServer (arrow 7) to perform the corresponding ptrace operations (arrow 8).

3.4. Debug client

We developed a simple command-line interface debug client to test the implementation of the debugger. It simply accepts inputs from a user and sends them to the debug service. The client is linked with GT4 client-side libraries and utilizes the security and authentication services of the middleware. A sample debugging session is given below.

```
donny@attica:~$ grid-proxy-info
...
path     : /tmp/x509up_u1000
timeleft : 99:19:29  (4.1 days)

donny@attica:~$ java org.globus.
monash.clients.DebugService.DebugClie
nt https://130.194.224.235:8443/wsrf/
services/monash/core/DebugFactoryServ
ice
Acme Debugger:
::> load /tmp/hello
::> waitForExecution
Inferior not connected yet
Inferior not connected yet
...
Inferior not connected yet
Ready to debug...
::> run
Debugger Out: Running
Program Out: Hello World
Program Out: Variable = 10
::> exit
Exiting...
donny@attica:~$
```

To use the Globus debug service, a user must have a valid Globus credential which is shown here by the grid-proxy-info command. The user loads the symbol file of a program and waits for a notification from the debug service. In another terminal, the user submits a job to WS-GRAM using the globusrun-ws command. The debug client is notified when the program is ready to be debugged (indicated by the "Ready to debug" message).

In addition, using a program such as Jython that integrates Python with Java, the debug service can be accessed by clients written in other languages as demonstrated below.

```
donny@attica:~$ jython
Jython 2.1 on java1.4.2-02 (JIT:
null)
Type "copyright", "credits" or
"license" for more information.
>>> import PyDebugClient
:: successfully loaded
>>> PyDebugClient.debug.debugInit
(PyDebugClient.DebugInit())
org.globus.monash.stubs.DebugService_
instance.DebugInitResponse@1
>>> PyDebugClient.debug.debugLoad
("/tmp/hello")
org.globus.monash.stubs.DebugService_
instance.DebugLoadResponse@1
>>> PyDebugClient.debug.debugBreak
point("main")
org.globus.monash.stubs.DebugService_
instance.DebugBreakpointResponse@1
>>> PyDebugClient.debug.debugRun
(PyDebugClient.DebugRun())
org.globus.monash.stubs.DebugService_
instance.DebugRunResponse@1
>>> PyDebugClient.debug.debugStepOver
(PyDebugClient.DebugStepOver())
org.globus.monash.stubs.DebugService_
instance.DebugStepOverResponse@1
>>> PyDebugClient.debug.debugPrint
("var")
'10'
>>>
```

4. Related work

There are several debuggers that are designed for testing and debugging Grid applications. Some examples of them are: p2d2 [3], the metadebugger in the Harness framework [5], Net-dbx-G [6], the Mercury Monitoring System [7], and PDB [8].

The Portable Parallel/Distributed Debugger (p2d2) is a project at the NASA Ames Research Center that developed a debugger for applications running on heterogeneous computational Grids [3]. It employs a client-server architecture and relies on GDB [4] as the low-level portable debugger. Instances of GDB communicate with a debug server which is implemented in C++ and maintains a collection of C++ objects to represent processes and stacks. The server is controlled by a

graphical user interface, the debug client, that allows users to examine the state and to control the execution of Grid applications. The current implementation of p2d2 only supports Globus Toolkit 2 jobs.

Harness is a metacomputing system that defines a simple but powerful architectural model to overcome the limited flexibility of traditional distributed software frameworks [5]. It consists of a kernel and plug-ins that provide various services for users. It is implemented in Java to leverage the homogenous architecture, the JVM, over heterogeneous computer platforms. Harness provides a distributed virtual machine for execution of metacomputing applications written in Java with Remote Method Invocations (RMI). A metadebugger has been developed in the Harness framework using the Java Platform Debug Architecture with remote debugging capability. The debugger is closely intertwined with the framework and it cannot be used with other Grid middleware.

Net-dbx-G is a web-based debugger for Message Passing Interface (MPI) programs executing on Grid resources [6]. It uses Java applets as the user interface and GDB [4] as the back-end debugger. It supports Globus Toolkit 2 (GT2) middleware with MPICH-G2 that provides the MPI programming library. Debugging an application requires the executable to be compiled with the Net-dbx-G instrumentation library. When the application is executed, an initialization method in the library notifies the debug client and spawns a child GDB process to debug the parent process. The user then controls GDB from a web browser using the Java applets.

The Mercury Monitoring System which is developed as part of the GridLab project is a generic Grid monitoring framework [7]. It provides support mainly for application monitoring, however, the recent release of Mercury allows users to perform remote debugging. It employs GDB and GDBServer [4]. GDBServer is used as the debug server on Grid nodes with GDB as the user interface on a programmer's desktop. Debugging is performed by sending a message to the Mercury monitoring library that is compiled into the application. The library then forks GDBServer and instructs it to attach to the debugged process. The developer controls the debug server using GDB from the desktop.

PDB is an implementation of the pervasive debugging approach [8]. It leverages the Xen Virtual Machine Monitor to virtualise the system resources used by a debugged application. The virtualisation allows a user to control and to inspect the complete state of the application and the resources including their low-level details such as processor instructions, system timers, and thread schedulers. By using Xen to virtualise Grid resources, users can deterministically debug Grid applications. However, this deterministic debugging technique is difficult to attain for applications that need to be run on a large-scale distributed system.

Different to all the debuggers described, our debugger is designed as a debug library with high-level application programming interfaces (API) suitable for debugging Grid applications. It has a layered and modular architecture that allows it to be plugged into any Grid middleware and to be used with any debug back-ends. Our approach simplifies the development process by not requiring the debugged applications to be compiled with an instrumentation/monitoring library.

5. Meeting the challenges

Rather than developing, yet again, a debugger for a particular Grid middleware, we have designed and implemented a modular library that can accommodate different Grid middleware and different debug back-ends. This design simplifies extending existing middleware with a debugging service and leverages existing debuggers. Furthermore, the library can be utilized by other tools. For example, profilers and high level tracers can be built on top the library. In addition, tools for enforcing software contracts (pre and post conditions) [16] for Grid applications could be implemented with ease by leveraging the APIs.

The current implementation of a Grid debugging service utilizes Globus Toolkit 4 as the middleware layer and GDB as the debug engine. A discussion on how this service meets the challenges listed in section 1 is given below.

Job scheduling. By implementing a callback mechanism in the service, the task of initiating a

debugging session is inherent in the way an application is initiated. This mechanism alleviates the need for a programmer to constantly poll the job scheduler to check whether the application has been started.

Hierarchical resources. The callback mechanism eliminates the need for the programmer to access intermediate resources (firewall server, gateway machine, or front node) to debug an application. The programmer is required, however, to wait for an incoming connection from the execution node where the debugged application is running and this is managed by the new framework.

Security. Our debugging service is implemented as a library that leverages the security and authentication services provided by Grid middleware. It does not require changes to the Grid security policy in place. In addition, since the debugging activity is initiated by the application rather than by the programmer, a user cannot arbitrarily debug another user's processes.

Heterogeneity. The issue of heterogeneity is addressed by proposing and implementing standard methods and API that could support a number of tools and middleware for Grid application debugging.

A debugger for programs running on heterogeneous architectures must support various data representations that occur because of different architectural features such as byte ordering and word length. A significant body of research has been conducted by our group on an architecture independent data format (AIF) [19][20][21]. When data is converted to AIF, it is labeled with a format descriptor string that describes the layout and the size of the data. AIF facilitates means for addressing machine heterogeneity in debugging Grid applications.

We plan to adopt the Grid debugging service described in this paper as part of our integrated framework for Grid application development [17]. In addition, because the library is modular, it could be adopted as the underlying debugging service for other Grid tools such as g-Eclipse [18].

6. Conclusion

This research paper presents the design of a Grid debugging service and an implementation of the service in the form of a WSRF-compliant debugger for Grid applications. A debug library with a set of well-defined debug API based on the High Performance Debugging Forum (HPDF) standard is also described. While a complete service using Globus Toolkit 4 and GDB has been implemented, further testing and implementation using other Grid middleware and debug back-ends need to be conducted.

7. References

[1] Foster, I., Kesselman, C. and Tuecke, S. (2001) The anatomy of the grid: enabling scalable virtual organizations. International Journal of Supercomputer Applications, 15, 200-222.

[2] Balle, S. M. and Hood, R. T., GGF UPDT User Development Tools Survey, March 2004.

[3] Hood, R. and Jost, G. A Debugger for Computational Grid Applications. Heterogeneous Computing Workshop 2000: 262-270.

[4] GDB: The GNU Debugger Home Page, http://www.gnu.org/software/gdb/gdb.html.

[5] Lovas, R. and Sunderam, V. Debugging of Metacomputing Applications.
Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS-JPDC), Fort Lauderdale, FL, USA, 2002.

[6] Neophytou, P., Neophytou, N., Evripidou, P. Net-dbx-g: A Web-based Debugger of MPI Programs Over Grid Environments, Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium, 19-22 April 2004, Pages:35 - 42.

[7] Gombas, G., Marosi, C. A., and Balaton, Z. Grid Application Monitoring and Debugging Using the Mercury Monitoring System. EGC 2005, European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005: 193-199.

[8] Mehmood, R., Crowcroft, J., Hand, S., and Smith, S. Grid-Level Computing Needs Pervasive Debugging. Proceedings of Grid 2005, 6th IEEE/ACM International Workshop on Grid Computing Seattle, Washington, USA, November 2005.

[9] High Performance Debugging Forum's HPD Version 1 Standard: Command Interface for Parallel Debuggers, (Rev. 2.1), 1998. http://www.ptools.org/hpdf/draft.

[10] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2005.
[11] Atkinson, M. Rationale for choosing the Open Grid Services Architecture, in Grid Computing: Making the Global Infrastructure a Reality, Wiley, 2003.
[12] The WS-Resource Framework, http://www.globus.org/wsrf/specs/ws-wsrf.pdf.
[13] Debugging with GDB, http://sources.redhat.com/gdb/current/onlinedocs/gdb.html.
[14] Joseph, J. and Fellenstein, C., Grid Computing, Prentice Hall, 2004.
[15] Francioni, J and Pancake, C., High Performance Debugging Standards Effort, http://web.engr.oregonstate.edu/~pancake/papers/HPDebugForum.pdf
[16] Meyer, B., Design by Contract, in Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer, Prentice Hall, 1991, pp. 1-50
[17] Kurniawan, D., Abramson, D., Worqbench: an Integrated Framework for e-Science Application Development. e-Science meeting 2006, Netherlands, December 2006.
[18] g-Eclipse – Access the Grid, http://www.geclipse.org.
[19] Abramson, D.A., Sosic, R. and Watson, G., Implementation Techniques for a Parallel Relative Debugger, International Conference on Parallel Architectures and Compilation Techniques - PACT '96, October 20-23, 1996, Boston, Massachusetts, USA.
[20] Watson, G. and Abramson, D., Relative Debugging For Data Parallel Programs: A ZPL Case Study, IEEE Concurrency, Vol 8, No 4, October 2000, pp 42 – 52.
[21] Abramson, D., Finkel, R., Kurniawan, D., Kowalenko, V. and Watson, G., Parallel Relative Debugging with Dynamic Data Structures, 16th International Conference on Parallel and Distributed Computing Systems, August 13 - 15, 2003 Reno, Nevada, USA.