A Lightweight Kernel for the Harness Metacomputing Framework *

C. Engelmann and G. A. Geist
Computer Science and Mathematics Division
Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
{engelmannc,gst}@ornl.gov
http://www.csm.ornl.gov

Abstract

Harness is a pluggable heterogeneous Distributed Virtual Machine (DVM) environment for parallel and distributed scientific computing. This paper describes recent improvements in the Harness kernel design. By using a lightweight approach and moving previously integrated system services into software modules, the software becomes more versatile and adaptable. This paper outlines these changes and explains the major Harness kernel components in more detail. A short overview is given of ongoing efforts in integrating RMIX, a dynamic heterogeneous reconfigurable communication framework, into the Harness environment as a new plug-in software module. We describe the overall impact of these changes and how they relate to other ongoing work.

1. Introduction

The heterogeneous adaptable reconfigurable networked systems (Harness [9]) research project is an ongoing collaborative effort among Oak Ridge National Laboratory, The University of Tennessee, Knoxville, and Emory University. It focuses on the design and development of a pluggable lightweight heterogeneous Distributed Virtual Machine (DVM) environment, where clusters of PCs, workstations, and "big iron" supercomputers can be aggregated to form one giant DVM (in the spirit of its widely-used predecessor, "Parallel Virtual Machine" (PVM) [10]).

As part of the Harness project, a variety of experiments and system prototypes were developed to explore lightweight pluggable frameworks, adaptive reconfigurable runtime environments, assembly of scientific applications from software modules, parallel plug-in paradigms, highly available DVMs, fault-tolerant message passing, fine-grain security mechanisms and heterogeneous reconfigurable communication frameworks.

Currently, there are three different Harness system prototypes, each concentrating on different research issues. The teams at Oak Ridge National Laboratory [3, 5, 12] and at the University of Tennessee [6, 7, 13] provide different C variants, while the team at Emory University [11, 14, 15, 18] maintains a Java-based alternative.

Conceptually, the Harness software architecture consists of two major parts: a runtime environment (kernel) and a set of plug-in software modules. The multi-threaded user-space kernel manages the set of dynamically loadable plug-ins. While the kernel provides only basic functions, plug-ins may provide a wide variety of services needed in fault-tolerant parallel and distributed scientific computing, such as messaging, scientific algorithms and resource management. Multiple kernels can be aggregated into a Distributed Virtual Machine.

This paper describes recent improvements in the kernel design, which significantly enhance versatility and adaptability by introducing a lightweight design approach. First, we outline the changes in the design and then explain the major kernel components in more detail. We continue with a short overview of recent efforts in integrating RMIX, a dynamic heterogeneous reconfigurable communication framework, into the Harness environment. We describe the overall impact of these changes and how they relate to other ongoing work. This paper concludes with a brief summary of the presented research.

2. Kernel Architecture

Earlier designs of a multi-threaded kernel [3] for Harness were derived from an integrated approach similar to PVM [10], since the Harness software was initially developed as a follow-on to PVM, based on the distributed virtual machine (DVM) model.

* This research is sponsored by the Mathematical, Information, and Computational Sciences Division; Office of Advanced Scientific Computing Research; U.S. Department of Energy. The work was performed at the Oak Ridge National Laboratory, which is managed by UT-Battelle, LLC under Contract No. De-AC05-00OR22725.

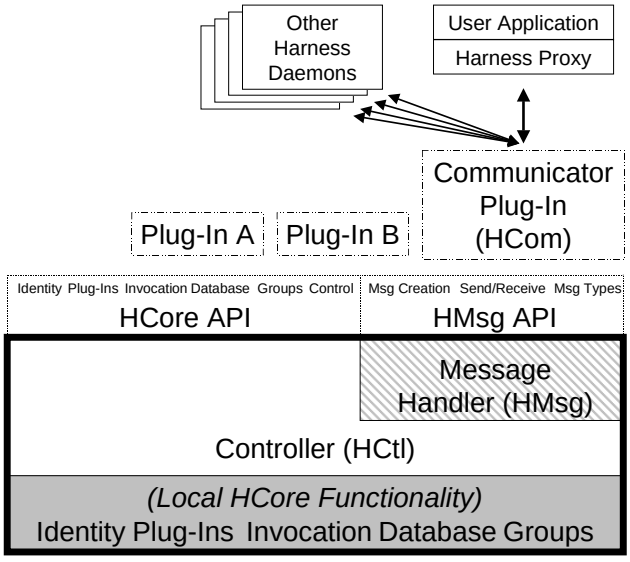

Figure 1. Previous Kernel Design

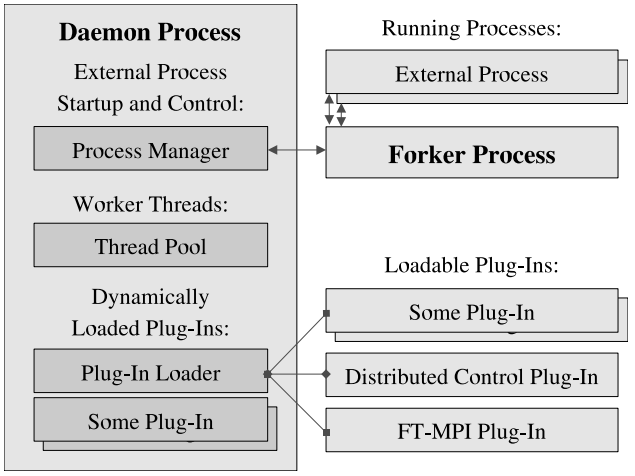

Figure 2. New Lightweight Kernel Design

In PVM, every node runs a local virtual machine and the overall set of virtual machines is controlled by a single master. This master is a single point of control and failure. In the DVM model, all nodes form together a *distributed* virtual machine, which they equally control in virtual synchrony. Symmetric state replication among all nodes, or a subset, assures high availability. Since there is no single point of control or failure, the DVM survives as long as at least one node is still alive.

The original Harness kernel design (Figure 1) featured all components needed to manage plug-ins and external processes, to send messages between components inside and

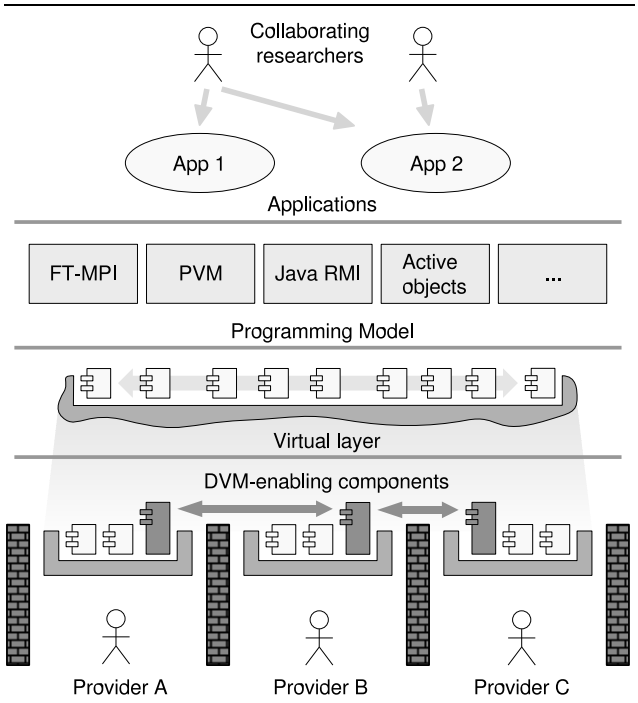

Figure 3. Harness Architecture

between kernels, and to control the DVM. Recent improvements in the design (Figure 2) are based on a lightweight approach. By moving previously integrated services into plug-ins, the kernel becomes more versatile and adaptable.

The new design features a lightweight kernel with plug-in, thread and process management only. All other basic components, such as communication, distributed control and event notification, are moved into plug-ins in order to make them reconfigurable and exchangeable at runtime. The kernel provides a container for the user to load and arrange all software components based on actual needs, without any preconditions imposed by the managing framework (Figure 3). The user can choose the programming model (peer-to-peer, client/server, PVM, MPI, Harness DVM) by loading the appropriate plug-in(s).

The kernel itself may run as a daemon process and accepts various command line and configuration file options, such as loading default plug-ins and spawning certain programs on startup, for bootstrapping with an initial set of services. A debug variant with more extensive error logging and memory tracing is also provided using the '.debug' extension or "–debug" command line flag.

In the following sections, we describe the three major kernel components (process manager, thread pool and plug-in loader) in more detail.

2.1. Process Manager

Harness is a pluggable multi-threaded environment. However, in heterogeneous distributed scientific computing scenarios it is sometimes necessary to fork a child process and execute a different program, e.g. for remote logins using the *ssh* program. The process manager allows just that. In addition, it also provides access to the standard input and output channels of spawned programs, allowing plug-ins to initiate and control them.

Only programs that reside in a specific preset program directory, or in a respective sub-directory, may be executed. Links to system programs, such as *ssh*, may be created by the system administrator.

Forking a process in a multi-threaded environment is not very well defined. On startup, the Harness kernel immediately creates a separate process that is responsible for launching child processes and relaying standard input and output between spawned programs and the kernel. This avoids any problems associated with duplicating threads and file descriptors in the kernel.

2.2. Thread Pool

Due to the potentially unpredictable runtime configuration of the Harness environment, certain issues regarding thread creation, termination and synchronization need to be addressed. They range from dealing with the limited number of threads controllable by a single process to stalling the system by using too many simultaneous threads. The thread pool allows the execution of jobs in threads, while staying in a preset range of running threads utilizing a job queue.

Jobs are submitted to the thread pool in the form of a job function pointer and a pointer to a job function argument. The thread pool always maintains a minimum number of threads to offer quick response times. The thread count is increased for every newly submitted job if no idle thread is available and a preset maximum is not reached. Jobs are queued into a fixed length job queue after reaching this maximum. Job submissions are finally blocked when the job queue is full. Idle threads time out and exit if the minimum number of threads is not yet reached.

Most thread pool properties, such as the maximum number of threads, are configured during kernel startup and may be modified via command line options and a kernel configuration file. They may also be modified at runtime via the thread pool configuration functions. This is essential for long running jobs which permanently occupy a thread, such as network servers. The maximum number of threads can be easily adjusted to avoid thread pool starvation.

2.3. Plug-in Loader

The plug-in loader handles loading and unloading of runtime plug-ins, which exist in the form of shared libraries. Such plug-ins are automatically unloaded on kernel shutdown. Plug-ins may load and unload other plug-ins on demand. Only plug-in modules that reside in a specific preset plug-in directory, or in a respective sub-directory, may be loaded. The kernel itself may load plug-ins on startup, in a separate thread that applies command line options.

Every plug-in has the opportunity to initialize itself during loading and to finalize itself during unloading via *<plug-in name>_init()* and *<plug-in name>_fini()* functions that are automatically resolved and called by the plug-in loader if they exist. Plug-ins are able to load and unload other plug-ins they depend on during initialization and finalization, since the plug-in loading and unloading functions are reentrant and deadlock free.

A plug-in (shared library) is loaded by the plug-in loader via *dlopen()* without exporting any of its symbols (functions and data) to the global name space of the kernel. Plug-in symbols are not made automatically available to the kernel or to other plug-ins, to prevent naming conflicts and to allow multiple plug-ins to offer the same interface with different implementations. In fact, only the module (kernel or another plug-in) that loads a plug-in may have access to it using a unique and private handle.

The plug-in loading and unloading policy is based on ownership. A module that loads a specific plug-in owns it and is also responsible for unloading it. The ownership of orphaned plug-ins is automatically transferred to the kernel. They are unloaded on kernel shutdown. The plug-in loader supports multiple loading to allow different modules to load the same plug-in. Plug-ins that maintain state may use reference counting and separate state instances to support multiple loading.

Plug-in symbols are resolved by the plug-in loader via *dlsym()*, requiring the plug-in handle and a symbol name. Plug-ins may provide tables with function and data pointers to improve performance by offering a one-time lookup. In this case, plug-in functions are called and plug-in data is accessed virtually similar to C++ objects.

The plug-in loader also supports a specific versioning scheme for plug-ins similar to the library versioning of the GNU Libtool [16]. A single number describes the plug-in API version and a single number describes the revision of this specific plug-in API implementation. The version number is increased and the revision number is set to 0 if the API was changed. The revision number is increased if the implementation of the API was improved, e.g. in case of bug fixes and performance enhancements.

A plug-in may also support past API versions if functions where only added, so that there is always a continuous

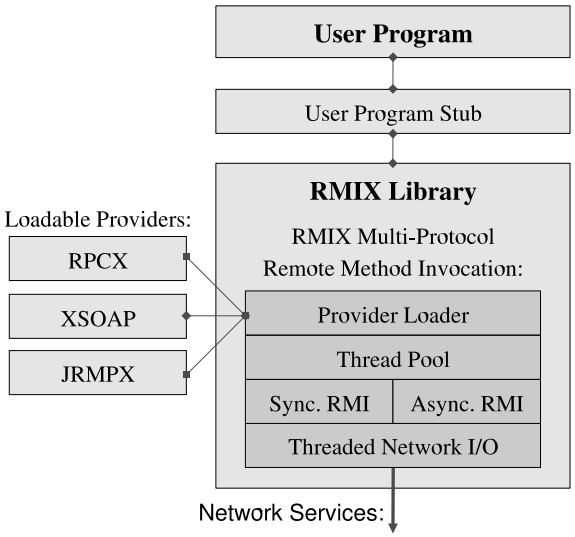

Figure 4. RMIX Framework Design

range of versions specified by two numbers, the first version and the number of follow-up versions (age). Together with the revision, the triplet <*first*>.<*age*>.<*revision*> is formed, which is used to differentiate and order files and folders by appending it to or inserting it into file or folder names, e.g. *libfoo.1.2.3.so*.

The plug-in loader allows one to specify the desired version in addition to the plug-in name when loading a plug-in. It chooses the most recent implementation of this version. Furthermore, age and revision may also be specified.

In order to provide more extensive debugging support, plug-in variants with built-in extended error logging and memory tracing are supported as well using the '.debug' extension, e.g. *libfoo.debug.so* or *libfoo.1.2.3.debug.so*. Debug plug-in variants are automatically preferred when loading into the kernel debug variant.

3. RMIX Framework

RMIX [15] is a dynamic heterogeneous reconfigurable communication framework, initially developed by the Harness team at Emory University, that allows Java applications to communicate via TCP/IP using various RMI/RPC protocols, like Sun RPC, Java RMI and SOAP. In addition to standard synchronous RMI/RPC mechanisms, RMIX also allows support for asynchronous and one-way invocations, which suits well the messaging needs of the peer-to-peer distributed control in Harness [5]. Ongoing research at Oak Ridge National Laboratory targets the development of a stand-alone C variant of RMIX and its integration into the lightweight Harness framework in order to replace the inflexible HCom communicator plug-in, thus improving adaptability and heterogeneity.

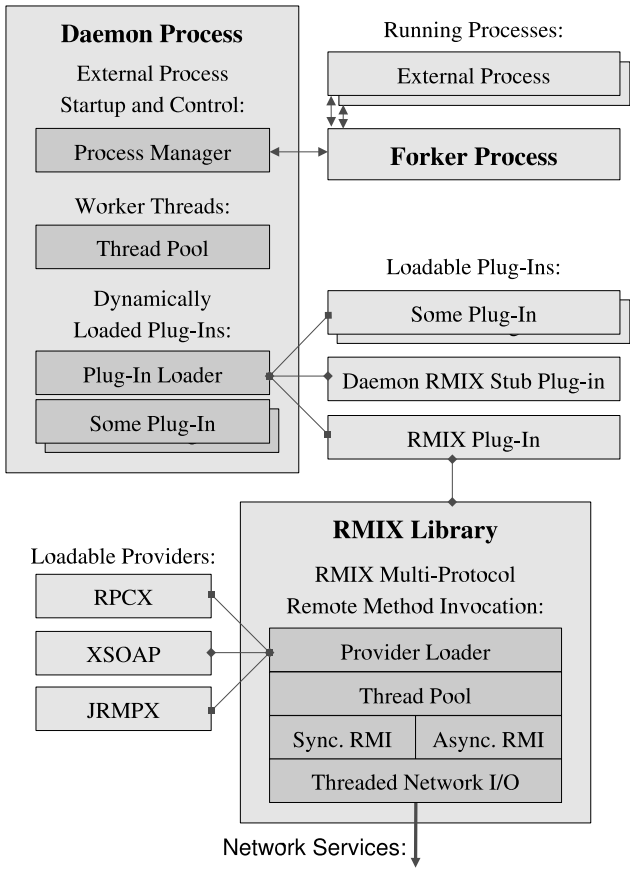

Figure 5. RMIX Plug-in for Harness Kernel

The C based RMIX framework (Figure 4) consists of a base library and a set of runtime provider plug-ins (shared libraries). The base library offers the same plug-in loader and thread pool as the Harness kernel. The provider plug-ins are responsible for the protocol as well as for the TCP/IP communication. In the future, provider plug-ins may also support different inter-process communication methods, e.g. pipes and shared memory.

The RMIX framework offers access to remote network services via client stubs, which provide the illusion of local invocation of remote methods. Client stub methods have the same signature as the remote methods they represent. They translate the method arguments into and the method return value from a generic representation that is being used by the provider plug-ins.

Network services are implemented using the RMIX framework via server stubs, which translate the method arguments from and the method return value into the generic representation. The same provider plug-ins are being used on the server side.

The RMIX framework is integrated into Harness as a

base plug-in and a stub plug-in (see Figure 5). The base plug-in is linked directly to the RMIX base library and provides its functions to other Harness plug-ins. The stub plug-in contains all client and server stubs necessary to invoke local or remote kernel functions.

On loading into the kernel, the RMIX stub plug-in automatically loads the RMIX base plug-in and optionally starts one or more RMI/RPC servers depending on a RMIX stub plug-in configuration file. Using the command line bootstrap mechanism, a kernel may provide all of its essential services via RMI/RPC servers at startup.

4. Impact

The described recent changes and improvements to the Harness lightweight kernel design were motivated by a lack of variety in programming paradigms. The previously embedded DVM programming paradigm forced a Harness user to run its application or service in the DVM context. However, more loosely coupled peer-to-peer paradigms that do not maintain global knowledge (state) among the participating processes where only supported through the DVM, thus introducing an additional overhead.

The main reasons for embedding the DVM into the kernel were the original PVM-like design approach and the requirement for high availability of the kernel itself. Everything that depends on the kernel can be highly available, since its managing framework is highly available. However, managing global state among a large number of nodes has scalability limits. Even with caching and partitioning hierarchies, the DVM approach is not suited for systems with tens or hundreds of thousands of processors, such as are on the horizon, e.g. the IBM BlueGene\L. Localized peer-to-peer concepts [4] can be much more efficient on such extreme scale systems.

With the new design, the DVM management, i.e. the distributed control [5], has been moved into a plug-in. The DVM plug-in now handles all DVM requests, such as loading a plug-in or replicating plug-in state for high availability. It also implements failure recovery and event notification. However, high availability is limited to plug-ins that rely on the DVM plug-in. The kernel itself is no longer highly available.

The lightweight design now allows plug-ins to exist outside of the DVM context. Furthermore, highly available plug-ins inside the DVM are able to use plug-ins outside of the DVM for simple tasks, which can be easily restarted during recovery. For example, a communication plug-in, such as HCom or RMIX, does not need to be highly available, since its service (providing communication endpoints) is bound to a specific location.

An early prototype of the new design showed significant improvements in usability, versatility and adaptability. Especially the capability for plug-in loading at kernel startup, without having a DVM running, proved to be very useful. For example, a worker task for a scientific calculation can be easily started by simply spawning the kernel on a remote machine using *ssh* with the appropriate application plug-in name in the kernel command line. The application plug-in has full access to its local kernel and may load the DVM plug-in for high availability and/or the HCom or RMIX plug-in to communicate with other kernels. It also may just perform some local computation and relay the result back to a repository.

In contrast, the previous Harness design required a user to start the kernel via a PVM-like command line tool on the local and subsequently remote machine, forming a DVM. The application plug-in was then loaded and accessed via the DVM management, replicating some or all of its state throughout the DVM server process group.

With our new design, we have experienced performance improvements in some cases, e.g. where kernel functions were used without going through the DVM management. The DVM related overhead depends on the number of participating nodes inside the DVM and the current load (pending requests) of the DVM management. Request processing times can vary from less than a second to a few seconds under normal conditions and up to several minutes in large systems under heavy load. This time can be saved when bypassing the DVM management for tasks that do not need to be highly available.

For example, applications that follow the bag-of-tasks programming paradigm can easily be made fault-tolerant by farming out tasks and restarting them at different locations on failure. Task farming with on-the-fly fault tolerance by task replacement is a widely used technique today. Examples are SETI@HOME [17] and Condor [1, 2]. The new kernel design efficiently supports this programming paradigm without the DVM overhead.

The kernel design changes also made it easier to program plug-ins. For example, the new thread pool gives the kernel sole control over thread management, rather than each individual plug-in. Plug-in programmers need only write their job functions and submit them to the thread pool. Thread cancellation clean-up on kernel shutdown is supported via the *pthread_cleanup* interface.

5. Related Work

This paper describes the recent changes and improvements to the C-based variant of the Harness kernel maintained at Oak Ridge National Laboratory. The Harness team at Emory University developed a Java-based version, H2O [18], with a similar design.

H2O is a secure, scalable, stateless, lightweight, flexible, resource sharing platform. It allows resource owners,

any authorized third parties or clients themselves to deploy services (pluglets) into the H2O container (kernel). H2O has been designed to support a wide range of distributed programming paradigms, including self-organizing applications, widely distributed applications, massively parallel applications, task farms and component composition frameworks. H2O is founded on the RMIX communication framework. Common usage scenarios involve clients deploying computational services just prior to using them, thus availing of the raw computational power shared by the container owner. Hence, resource sharing and grid computing systems can naturally be formed using H2O.

Our Harness kernel and H2O have similar lightweight designs. However, H2O is far more advanced in terms of security mechanisms, while it does not fully support native plug-ins (shared libraries). This is due to the fact that the Java Native Interface does not allow multiple Java objects to simultaneously load the same native plug-in.

The Harness team at the University of Tennessee (UTK) also maintains a C-based variant which has a lightweight design. However, it uses a ring-based replicated database for high-availability, where our solution is based on more advanced distributed control [5]. The Harness variant from UTK also provides a PVM-like console control program to manage DVMs. Furthermore, UTK developed an FT-MPI [7, 6, 8] plug-in for fault-tolerant MPI messaging in Harness.

The ongoing research effort in Harness includes the cooperative integration of technology between members of the overall Harness team. Currently, FT-MPI is being combined with H2O, for support across administrative boundaries. RMIX is being implemented in C as a stand-alone solution, as well as a native Harness plug-in, to provide flexible heterogeneous communication for all Harness frameworks, and for any other heterogeneous distributed computing platforms.

6. Conclusions

We have described recent changes and improvements to the Harness lightweight kernel design. By using a more efficient and flexible approach and moving previously integrated services into distinct plug-in modules, the software becomes more versatile and adaptable. The kernel provides a container for the user to load and arrange all software components based on actual needs, without any preconditions imposed by the managing framework. The user chooses the programming model (DVM, PVM, MPI, etc.) by loading the appropriate plug-in(s).

Recent efforts were described in which RMIX, a dynamic heterogeneous reconfigurable communication framework, was integrated into the Harness environment as a new pluggable software module.

Our experience with an early prototype has shown that the new kernel design is able to efficiently support multiple programming models without additional DVM overhead, simply by bypassing the DVM management for tasks that do not need high availability. Furthermore, the design changes have also made it easier to program plug-ins.

Future work will concentrate on providing service-level high availability features to applications, as well as to typical operating system components, such as schedulers.

References

[1] J. Basney and M. Livny. Deploying a high throughput computing cluster. In R. Buyya, editor, *High Performance Cluster Computing: Architectures and Systems, Volume 1*. Prentice Hall PTR, 1999.

[2] Condor at University of Wisconsin, Madison, WI, USA. At http://www.cs.wisc.edu/condor.

[3] W. R. Elwasif, D. E. Bernholdt, J. A. Kohl, and G. A. Geist. An architecture for a multi-threaded Harness kernel. *Lecture Notes in Computer Science: Proceedings of PVM/MPI User's Group Meeting 2001*, 2131:126–134, 2001.

[4] C. Engelmann and A. Geist. A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform. *Proceedings of CLADE 2003*, pages 47–52, 2003.

[5] C. Engelmann, S. L. Scott, and G. A. Geist. Distributed peer-to-peer control in Harness. *Lecture Notes in Computer Science: Proceedings of ICCS 2002*, 2330:720–728, 2002.

[6] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. Harness and fault tolerant MPI. *Parallel Computing*, 27(11):1479–1495, 2001.

[7] G. E. Fagg, A. Bukovsky, S. Vadhiyar, and J. J. Dongarra. Fault-tolerant MPI for the Harness metacomputing system. *Lecture Notes in Computer Science: Proceedings of ICCS 2001*, 2073:355–366, 2001.

[8] FT-MPI at University of Tennessee, Knoxville, TN, USA. At http://icl.cs.utk.edu/ftmpi.

[9] G. Geist, J. Kohl, S. Scott, and P. Papadopoulos. HARNESS: Adaptable virtual machine environment for heterogeneous clusters. *Parallel Processing Letters*, 9(2):253–273, 1999.

[10] G. A. Geist, A. Beguelin, J. J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, MA, USA, 1994.

[11] Harness at Emory University, Atlanta, GA, USA. At http://www.mathcs.emory.edu/harness.

[12] Harness at Oak Ridge National Laboratory, TN, USA. At http://www.csm.ornl.gov/harness.

[13] Harness at University of Tennessee, Knoxville, TN, USA. At http://icl.cs.utk.edu/harness.

[14] D. Kurzyniec, V. S. Sunderam, and M. Migliardi. PVM emulation in the Harness metacomputing framework - Design and performance evaluation. *Proceedings of CCGRID 2002*, pages 282–283, 2002.

[15] D. Kurzyniec, T. Wrzosek, V. Sunderam, and A. Slominski. RMIX: A multiprotocol RMI framework for Java. *Proceedings of IPDPS 2003*, pages 140–145, 2003.

[16] GNU Libtool - The GNU portable library tool. At http://www.gnu.org/software/libtool.

[17] SETI@HOME at University of California, Berkeley, CA, USA. At http://setiathome.ssl.berkeley.edu.

[18] V. Sunderam and D. Kurzyniec. Lightweight self-organizing frameworks for metacomputing. *Proceedings of HPDC 2002*, pages 113–124, 2002.

Biographies

Christian Engelmann is a Research and Development staff member in the Network and Cluster Computing Group at Oak Ridge National Laboratory (ORNL). Christian is also a postgraduate research student at the Department of Computer Science of the University of Reading.

He received his Advanced European M.Sc. degree in parallel and scientific computation from the University of Reading and his German Certified Engineer diploma in computer systems engineering from the College for Engineering and Economics (FHTW) Berlin in 2001. Christian joined ORNL in 2000 as a Master student.

He is a contributor to the Harness project, which focuses on developing a pluggable, lightweight, heterogeneous Distributed Virtual Machine environment. He is also a member of the MOLAR research team, that concentrates on developing adaptive, reliable, and efficient operating and runtime system solutions for ultra-scale high-end scientific computing. Furthermore, he is also involved with a project developing self-adapting, fault tolerant algorithms for systems with tens or even hundreds of thousands of processors. You can find out more about Christian on his web page: http://www.csm.ornl.gov/∼engelman

Al Geist is a Corporate Research Fellow at Oak Ridge National Laboratory (ORNL), where he leads the 35 member Computer Science Research Group. He is one of the original developers of PVM (Parallel Virtual Machine), which became a world-wide de facto standard for heterogeneous distributed computing.

Al was actively involved in both the MPI-1 and MPI-2 design teams and more recently the development of FT-MPI, a fault tolerant MPI implementation. Today He leads a national Scalable Systems Software effort, involving all the DOE and NSF supercomputer sites, with the goal of defining standardized interfaces between system software components. Al is co-PI of a national Genomes to Life (GTL) center. The goal of his GTL center is to develop new algorithms, and computational infrastructure for understanding protein machines and regulatory pathways in cells. He heads up a project developing self-adapting, fault tolerant algorithms for 100,000 processor systems.

In his 20 years at ORNL, he has published two books and over 190 papers in areas ranging from heterogeneous distributed computing, numerical linear algebra, parallel computing, collaboration technologies, solar energy, materials science, biology, and solid state physics. You can find out more about Al and a complete list of his publications on his web page: http://www.csm.ornl.gov/∼geist