An Architecture for a Multi-Threaded Harness Kernel [⋆]

Wael R. Elwasif, David E. Bernholdt, James A. Kohl, and G. A. Geist

Computer Science and Mathematics Division, Oak Ridge National Lab
Oak Ridge, TN 37831, USA
{elwasifwr,bernholdtde,gst,kohlja}@ornl.gov

Abstract. Harness is a reconfigurable, heterogeneous distributed metacomputing framework for the dynamic configuration of distributed virtual machines, through the use of parallel ”plug-in” software components. A parallel plug-in is a software module that exists as a synchronized collection of traditional plug-ins distributed across a parallel set of resources. As a follow-on to PVM, the Harness kernel provides a base set of services that plug-ins can use to dynamically define the behavior of the encompassing virtual machine. In this paper, we describe the design and implementation details of an efficient, multi-threaded Harness core framework, written in C. We discuss the rationale and details of the base kernel components – for communication, message handling, distributed control, groups, data tables, and plug-in maintenance and function execution – and how they can be used in the construction of highly dynamic distributed virtual machines.

Keywords: Harness, Parallel Plug-ins, PVM, Virtual Machines, Multi-threaded

1 Introduction

Next-generation high-performance scientific simulations will likely depart from present day parallel programming paradigms. Rather than relying on monolithic platforms that export a large, static operating environment, applications will need the flexibility to dynamically adapt to changing functional and computational needs. The application itself will dictate the ongoing configuration of the parallel environment, and will customize the run-time subsystems to suit its varying levels of interaction, roaming connectivity, and distinct execution phases.

Harness [2, 4, 9] is a heterogeneous distributed computing environment, developed as a follow-on to PVM [8]. Harness is a dynamically configurable distributed operating environment for parallel computing. It is designed with a lightweight kernel as a substrate for “plugging in” necessary system modules based on runtime application directives. Plug-ins can be coordinated in parallel across many networked machines, and can be dynamically swapped out during runtime to customize the capabilities of the system. Plug-in components can control all aspects of a virtual parallel environment,

[⋆] This material is based upon work supported by the Department of Energy under grant DE-FC0299ER25396, and the Mathematics, Information and Computational Sciences Office, Office of Advanced Scientific Computing Research, Office of Science, U. S. Department of Energy, under contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

including use of various network interfaces, resource or task management protocols, software libraries, and programming models (including PVM, MPI and others).

Harness is a cooperative effort among Oak Ridge National Laboratory, the University of Tennessee, and Emory University, and involves a variety of system prototypes and experiments that explore the nature and utility of parallel plug-ins and dynamically configured operating environments. This paper describes the architecture and implementation of an efficient, multi-threaded Harness kernel. This kernel exports the core interface for interacting with the Harness system, and provides a platform for the development of applications and plug-ins for use with Harness. The core functions provided by the kernel can be utilized to control the loading and unloading of plug-ins, the execution of arbitrary functions exported by plug-ins, and manipulation of system state information via a simple database interface.

While the merit of a "pluggable" design has been recognized in web browsers and desktop productivity suites, it has yet to manifest itself in collaborative heterogeneous computing environments for distributed scientific computing. PVM supports some limited dynamic pluggability, with "hoster", "tasker", and "resource manager" tools that can be instantiated at runtime to take over handling of certain system functions [10]. Traditional distributed computing platforms usually provide a static model in which the set of services available to applications cannot be easily altered. Globus [6] provides an integrated set of tools and libraries for accessing computational Grid resources. Legion [11] provides an object-based interface for utilizing arbitrary collections of computers across the global internet. NetSolve [3] consists of a front-end library and back-end server for farming out scientific computations on resources available over the network. The Harness model is distinct from these approaches in that it centers on the concept of the dynamic virtual machine as pioneered by PVM. User-level plug-ins can be added to Harness without modification to the base system, and Harness supports interactive loading and unloading of pluggable features during execution. The Harness (and PVM) virtual machine model provides flexible encapsulation and organization of resources and application tasks, allowing groups of computers and tasks to be manipulated as single unified entities.

Harness offers a perfect testbed for high-performance component-based systems such as the Common Component Architecture (CCA) [1, 7]. Both CCA and Harness build applications and system functionality using pluggable modules. Harness incorporates the dynamic capability to load and unload these components, where the CCA does not require this flexibility. For Harness to be a CCA-compliant framework, it need only load an appropriate "CCA Services" plug-in that implements the base CCA interfaces.

2 The Architecture of the Harness Daemon

The Harness daemon is a multi-threaded application that provides a runtime environment under which threadsafe plug-in functions run. The Harness daemon described here is being implemented in C-language at ORNL. C was chosen for this prototype to provide a high performance implementation with a small application footprint for more efficient use of resources. This design choice requires special handling that is implicitly taken care of in object-based languages, such as plug-in instantiation and function interface description. A Java-based prototype of Harness [14] has been under development at Emory University for the past year. Our use of C minimizes the runtime resource requirements, while still guaranteeing a high degree of portability in the production system, though with some increased complexity. Our resulting system architecture and interface are compatible with all source language implementations of Harness, given appropriate syntactic variations. Work such as the SIDL (Scientific Interface Description Language) [13, 15], as a cooperative project to the CCA, will likely provide the necessary language-specific processing to handle such variations.

The Harness daemon, in and of itself, is not intended to be a high-performance messaging system, nor do we expect many applications to make direct calls to the Harness APIs. The basic Harness APIs are intended primarily for use by plug-ins, which in turn provide the bulk of the actual user-level functionality. The services at the core of Harness are intended to serve two basic purposes: to provide the means for Harness daemons to exchange command and control information, and to act as a substrate for the loading/unloading and invocation of plug-in functionality. The design is intentionally minimalist to remain flexible and avoid unnecessary constraints on what plug-ins or user applications can do.

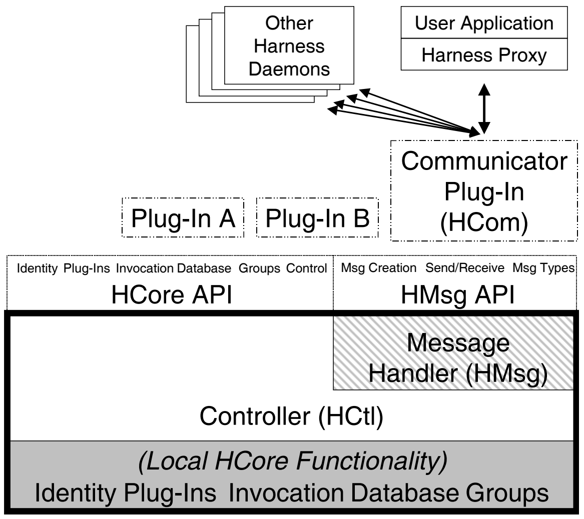
Other Harness Daemons
User Application
Harness Proxy
Communicator Plug-In (HCom)
Plug-In A
Plug-In B
Identity Plug-Ins Invocation Database Groups Control
HCore API
Msg Creation Send/Receive Msg Types
HMsg API
Message Handler (HMsg)
Controller (HCtl)
(Local HCore Functionality)
Identity Plug-Ins Invocation Database Groups

Fig. 1. A Schematic Representation of a Harness Daemon.

Figure 1 depicts a Harness daemon with several plug-ins, connected to a user application and to several other Harness daemons. The core of the Harness daemon, represented as the heavy box, contains three functional units with distinct roles: the local HCore functionality, the Controller (HCtl), and the Message Handler (HMsg). The HCore API functions are not directly exposed to the Harness user (e.g. the plug-in writer). All calls are routed through the Controller (HCtl) unit, which validates commands and coordinates their execution with the overall Harness virtual machine. The Message Handler (HMsg) unit exposes a second, self-contained API for processing and routing all incoming and outgoing messages. The HCore and HMsg APIs are exported for use by plug-ins and applications, represented in Figure 1 as dashed boxes.

The interactions between the HCtl and HMsg units are simple and well-defined, so modular implementations of each can be exchanged without impacting the other. These are not plug-ins, however, because *some* implementation of each unit must always be present, and they cannot be dynamically replaced. Yet this modularity provides an important flexibility, particularly with respect to the Controller.

2.1 The Controller

The Controller mediates the execution of all user-level calls, satisfying security requirements and insuring that all local actions are appropriately coordinated with the rest of the Harness virtual machine. Various choices of control algorithms can be imagined in different circumstances. Strictly local control, ignoring the existence of other Harness daemons, can be useful in a context where Harness is integrating several plug-in functionalities on a single host (e.g. performance monitoring, debugging, resource management). A master/slave algorithm could be used to mimic the behavior and performance characteristics of PVM. Perhaps the most interesting is a distributed, peer-based controller using an algorithm such as [5] to provide fault tolerance and recovery capabilities. Such algorithms involve multiple daemons working together to arbitrate commands and store the state information for the virtual machine. This eliminates the single point of failure in a master/slave control algorithm.

Within each of these broad classes of control algorithms, there are many possible variations in the details of how particular commands behave within the collective environment. The Controller *can* provide its own interface to allow tuning or modification of its behavior, likely via configuration parameters passed through the environment or a configuration file. This mechanism will be implementation-specific and is not considered part of the main Harness APIs.

2.2 The Local HCore Functionality

The HCore API functions can be divided into six distinct encapsulated modules: *Identity*, *Plug-Ins*, *Invocation*, *Database*, *Groups* and *Control*.[1] Each Harness daemon and application task has an opaque Harness ID, or HID, which is locally generated and globally unique. This HID is returned by `h_myhid()` and is used in many other functions. Harness provides functions to load and unload plug-ins and allow them to register exported methods. Function pointers to these plug-in methods can be acquired locally for direct invocation, or remote invocations can constructed and executed in blocking or non-blocking fashion. All registered plug-in functions are expected to accept an `H_arg` (essentially C's `argv`) as input. Functions in libraries or other software modules that export non-`H_arg` arguments can easily be converted to Harness plug-in functions by enclosing them in simple argument-parsing function wrappers.

[1] The full API for these functions is published on the Harness web site [12].

A simple database interface is available both to the Harness core and to plug-ins, and is the fundamental subsystem that the local Harness daemon and the virtual machine use to maintain their state. Database tables can be public or private. An `h_notify` function makes it possible for plug-ins to request notification on certain events which manifest themselves as changes to database tables.

Nearly all HCore functions take a “group” argument, specifying a subset of the current Harness virtual machine on which the functions are to be executed. Three groups are always available: *local*, *global* and *control*. The *local* and *global* groups indicate that the operation should take place on the local daemon or on *all* daemons participating in the virtual machine, respectively. The *control* group designates use of the selected decision-making daemon(s) in whatever control algorithm is running. Knowledge of this group is needed, for example, to allow “leaf” daemons (i.e. those daemons not directly participating in the decision-making control algorithm) to recover from faults or partitioning of the virtual machine. Users can also create other groups for their convenience through functions provided in the API.

Currently the only control function in the API (aside from any exported HCtl interface) is `h_exit()`, which terminates the local Harness daemon.

2.3 The Message Handler and Communication Modules

The core message processing of the Harness daemon is handled by two distinct but closely cooperating units: communication plug-ins and the message handler. Communication (HCom) units are plug-ins that are responsible for sending and receiving messages. HComs are not mandatory – it is possible to use Harness with strictly local functionality. However, most interesting uses of Harness involve inter-machine communication and therefore HComs. HComs treat message traffic as opaque, and are only responsible for putting messages out on the wire and receiving them from other nodes. Simultaneous use of multiple HCom plug-ins is possible. We expect that each HCom will handle all traffic of a given protocol, so that one might process TCP traffic, and another, for example, transmissions using the SOAP protocol.

The Message Handler (HMsg) unit is part of the daemon itself, and provides a uniform interface between the communication plug-ins and the daemon processing. HMsg is responsible for accepting outbound messages from local plug-ins, or from the HCore, and passing them on to HCom for transmission. HMsg is also responsible for processing of all inbound messages. As shown in Figure 2, HComs place inbound messages on a queue in HMsg. Within HMsg, messages are removed from the queue and routed according to their message type. Two pre-defined message types designate messages containing HCore interface commands and communications for the control algorithm. Plug-ins can register additional message types, and provide their own message handler functions. Aside from HCore messages, all messages are dispatched from the HMsg router directly to HCtl or to the appropriate registered plug-in handler. HCore command messages remain in HMsg for further processing and are parsed to extract the command name and arguments, which are then queued for processing by the local thread pool. This thread pool sits at the boundary of HMsg and HCtl, as HCore commands are executed only by passing them to HCtl for validation. HCtl passes commands on to the internal HCore layer to actually be carried out.

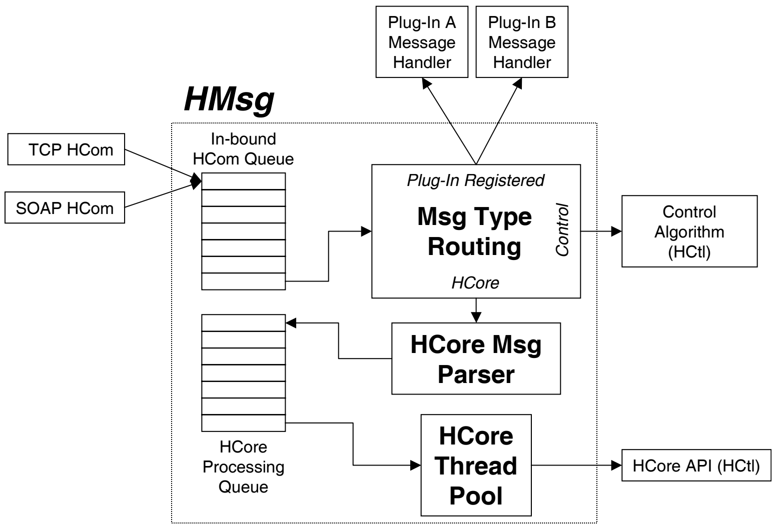

Fig. 2. Internal Message Handler Organization (HMsg).

HMsg provides an API for the construction and parsing of HCore command messages, and assumes symmetric and compatible HMsg implementations on both ends of any HCom messaging connection. Plug-ins can also provide their own functions to create and parse messages (for example those with their own message types). HMsg contains a user-level function to actually send the resulting message buffers. HMsg's role here is a small but crucial one: it determines which of the several possible HComs should be used to transfer the message, and then places the message on the outbound queue for that HCom.

The HCom/HMsg system is designed to be a flexible, low-overhead mechanism for the processing of messages. However we expect that high performance communication plug-ins will be developed and utilized for actual production scale data transfer in application tasks. The Harness infrastructure will be applied to set up customized *out-of-band* data channels. The internal Harness communication paths should be used for command and control functions but need not be used for actual data transmission.

3 Multi-Threaded Architecture

The Harness kernel is designed to be a multi-threaded application. Plug-in and HCore functions are executed in separate threads that are allocated from a worker pool managed by the kernel. In addition to these worker threads, plug-ins and other kernel components can also have multiple threads of execution. For example, the communication plug-in HCom can have one thread to handle system-level communication calls, and one or more threads to manage the interface to other components of the Harness kernel (e.g. the output queue fed by HMsg).

Using a multi-threaded Harness kernel protects the daemon from blocking unnecessarily, such as when a plug-in function waits for I/O. It is therefore not necessary to restrict plug-in functions to be non-blocking and return control immediately to the Harness kernel. The basic design philosophy of Harness is to afford plug-ins maximum operational flexibility. Plug-ins should be allowed to exploit blocking and/or non-blocking communication modalities, independent of the native kernel implementation. In addition, providing separate threads of execution within the Harness kernel enables "long-running" plug-in functions, such as those typically found in parallel applications which loop indefinitely in I/O-bound states. Separating these invocations leaves the Harness kernel available to process other requests. While much of this same functionality can be accomplished by instantiating each plug-in in its own process, this design incurs extra overhead for inter-plug-in communication and synchronization. Such issues are likely for cooperating parallel plug-in systems, and overheads could be significant for closely-coupled plug-ins.

As a result of the multi-threaded nature of the Harness kernel, it is necessary to require that all plug-in functions are threadsafe. To alleviate the potential complexity in converting non-threadsafe libraries into plug-ins, Harness provides a utility suite that includes basic threading synchronization operations and "mutexes" that assist plug-in functions in protecting critical sections and handling mutual exclusions. A trivial solution for a given library is to acquire a single mutex lock before executing *any* of its plug-in functions, however more complex (and efficient) solutions are possible using a set of distinct mutexes for dependent function groups. For plug-ins that cannot easily be made threadsafe through the use of simple mutex wrapper functions, a threadsafe *plug-in proxy* can coordinate plug-in function invocations with an external process that implements the given functions. The protocol by which such proxies would communicate is not dictated by Harness. As a side benefit, the Harness utility interface also serves as an abstraction layer that hides the implementation-specific details of the underlying thread implementation on a given platform (e.g. pthreads in Unix-based systems and Windows threads in Win32 systems).

4 Conclusion

We have described the multi-threaded architecture of the Harness kernel. Harness is being developed as a follow-on to PVM to explore research issues in reconfigurable heterogeneous distributed computing. Harness daemons are intended to provide basic command and control communication among themselves, and a substrate for dynamically plugging in user or system modules. These plug-ins are expected to be the main source of user-level functionality.

The Harness kernel presents two APIs to the user (plug-in developer). The HCore API provides the basic functionality of the daemon itself: managing plug-ins, invoking functions, and maintaining the database (which allows both the daemon and plug-ins to store state information). The HMsg API has a concise interface focused on the processing and sending of messages to other Harness tasks. The HMsg unit of the kernel, together with specialized HCom communication plug-ins, is responsible for all communication among Harness daemons, and provides a means for point-to-point communication among plug-ins as well. We have described the structure of the HMsg unit in some detail, as well as the multi-threaded nature of the Harness kernel.

References

1. Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, , and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *Proceedings of the The Eighth IEEE International Symposium on High Performance Distributed Computing*, 1998.
2. M. Beck, J. Dongarra, G. Fagg, A. Geist, P. Gray, J. Kohl, M. Migliardi, K. Moore, T. Moore, P. Papadopoulos, S. Scott, and V. Sunderam. HARNESS: a next generation distributed virtual machine. *Special Issue on Metacomputing, Future Generation Computer Systems*, 15(5/6), 1999.
3. Henri Casanova and Jack Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.
4. J. Dongarra, G. Fagg, A. Geist, and J. A. Kohl. HARNESS: Heterogeneous adaptable reconfigurable NEtworked systems. In IEEE, editor, *Proceedings: the Seventh IEEE International Symposium on High Performance Distributed Computing, July 28–31, 1998, Chicago, Illinois*, pages 358–359. IEEE Computer Society Press, 1998.
5. Christian Engelmann. *Distributed Peer-to-Peer Control for Harness.* M.Sc. thesis, University of Reading, 2001.
6. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.
7. Dennis Gannon, Randall Bramley, Thomas Stuckey, Juan Villacis, Jayashree Balasubramanjian, Esra Akman, Fabian Breg, Shridhar Diwan, and Madhusudhan Govindaraju. Developing component architectures for distributed scientific problem solving. *IEEE Computational Science & Engineering*, 5(2):50–63, April/June 1998.
8. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.
9. G. A. Geist. Harness: The next generation beyond PVM. *Lecture Notes in Computer Science*, 1497:74–82, 1998.
10. G. A. Geist, J. A. Kohl, P. M. Papadopoulos, and S. L. Scott. Beyond PVM 3.4: What we've learned, what's next, and why. *Special Issue on Metacomputing, Future Generation Computer Systems*, 15(5/6):571–582, 1999.
11. Andrew S. Grimshaw, William A. Wulf, and the Legion team. The legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.
12. ORNL Harness Home Page. `http://www.csm.ornl.gov/harness/`
13. S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. Divorcing language dependencies from a scientific software library. In *Proceedings, 10th SIAM Conference on Parallel Processing*, March 1999.
14. M. Migliardi and V. Sunderam. Plug-ins, layered services and behavioral objects application programming styles in the harness metacomputing system. *Future Generation Computer Systems*, 17:795–811, 2001.
15. B. Smolinski, S. Kohn, N. Elliott, N. Dykman, and G. Kumfert. Language interoperability for high-performance parallel scientific components. In *Proceedings, Int'l Sym. on Computing in Object-Oriented Parallel Environments (ISCOPE '99*, 1999.