Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System

Mauro Migliardi[1], Jack Dongarra[2,3], Al Geist[2], Vaidy Sunderam[1]

Emory University[1], Dept. Of Math & Computer Science
Atlanta, GA, 30322, USA
om@mathcs.emory.edu
Oak Ridge Natonal Laboratory[2],
University of Tennessee at Knoxville[3]

Abstract. Metacomputing frameworks have received renewed attention of late, fueled both by advances in hardware and networking, and by novel concepts such as computational grids. However these frameworks are often inflexible, and force the application into a fixed environment rather than trying to adapt to the application's needs. Harness is an experimental metacomputing system based upon the principle of dynamic reconfigurability not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via a "plug-in" mechanism that is the central feature of the system. In this paper we describe how the design of the Harness system allows the dynamic configuration and reconfiguration of virtual machines, including naming and addressing methods, as well as plug-in location, loading, validation, and synchronization methods.

1 Introduction

Harness is an experimental metacomputing system based upon the principle of dynamically reconfigurable networked computing frameworks. Harness supports reconfiguration not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via a "plug-in" mechanism that is the central feature of the system. The motivation for a plugin-based approach to reconfigurable virtual machines is derived from two observations. First, distributed and cluster computing technologies change often in response to new machine capabilities, interconnection network types, protocols, and application requirements. For example, the availability of Myrinet [1] interfaces and Illinois Fast Messages has recently led to new models for closely coupled Network Of Workstations (NOW) computing systems. Similarly, multicast protocols and better algorithms for video and audio codecs have led to a number of projects that focus on tele-presence over distributed systems. In these instances, the underlying middleware either needs to be changed or re-constructed, thereby increasing the effort level involved and hampering interoperability. A virtual machine model intrinsically incorporating reconfiguration capabilities will address these issues in an effective manner. The second reason for investigating the plug-in model is to attempt to provide a virtual machine environment that can dynamically adapt to meet an application's needs, rather than forcing the application to fit into a fixed environment. Long-lived simulations evolve through several phases: data input, problem setup, calculation, and analysis or visualization of results. In traditional, statically configured metacomputers, resources needed during one phase are often underutilized in other phases. By allowing applications to dynamically reconfigure the system, the overall utilization of the computing infrastructure can be enhanced.

The overall goals of the Harness project are to investigate and develop three key capabilities within the framework of a heterogeneous computing environment:

- Techniques and methods for creating an environment where multiple distributed virtual machines can collaborate, merge or split. This will extend the current network and cluster computing model to include multiple distributed virtual machines with multiple users, thereby enabling standalone as well as collaborative metacomputing.
- Specification and design of plug-in interfaces to allow dynamic extensions to a distributed virtual machine. This aspect involves the development of a generalized plug-in paradigm for distributed virtual machines that allows users or applications to dynamically customize, adapt, and extend the distributed computing environment's features to match their needs.
- Methodologies for distinct parallel applications to discover each other, dynamically attach, collaborate, and cleanly detach. We envision that this capability will be enabled by the creation of a framework that will integrate discovery services with an API that defines attachment and detachment protocols between heterogeneous, distributed applications.

In the preliminary stage of the Harness project, we have focused upon the dynamic configuration and reconfiguration of virtual machines, including naming and addressing schemes, as well as plugin location, loading, validation, and synchronization methods. Our design choices, as well as the analysis and justifications thereof, and preliminary experiences, are reported in this paper.

2 Architectural Overview of Harness

The architecture of the Harness system is designed to maximize expandability and openness. In order to accommodate these requirements, the system design focuses on two major aspects:

- the management of the status of a Virtual Machine that is composed of a dynamically changeable set of hosts;
- the capability of expanding the set of services delivered to users by means of plugging into the system new, possibly user defined, modules on-demand without compromising the consistency of the programming environment.

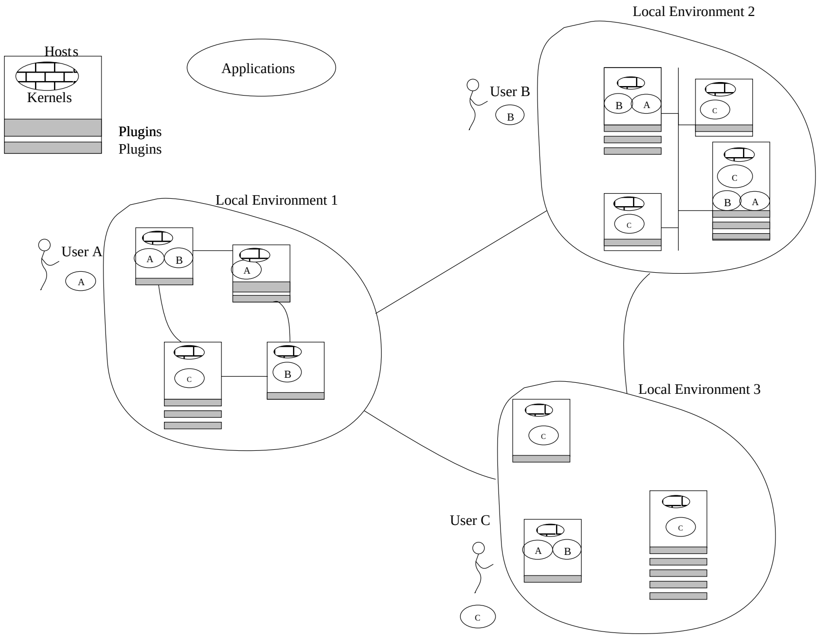

Fig. 1. A Harness Virtual Machine

2.1 Virtual Machine Startup and Harness System Requirements

The Harness system allows the definition and establishment of one or more Virtual Machines (VMs). A Harness VM (see figure 1) is a distributed system composed of a VM status server and a set of kernels running on hosts and delivering services to users.

The current prototype of the Harness system implements both the kernel and the VM status server as pure Java programs. We have used the multithreading capability of the Java Virtual Machine to exploit the intrinsic parallelism of the different tasks the programs have to perform, and we have built the system as a package of several Java classes. Thus, in order to be able to use the Harness system a host should be capable of running Java programs (i.e. must be JVM equipped). The different components of the Harness system communicates through reliable unicast channels and unreliable multicast channels. In the current prototype these communication commodities are implemented using the java.net package.

In order to use the Harness system, applications should link to the Harness core library. The basic Harness distribution will include core library versions for C, C++ and Java programs but in the following description we show only Java prototypes.

This library provides access to the only hardcoded service access point of the

Harness system, namely the core function

```
Object H_command(String VMSymbolicName, String[] theCommand).
```

The first argument to this function is a string specifying the symbolic name of the virtual machine the application wants to interact with. The second argument is the actual command and its parameters. The command might be one of the User Kernel Interface commands as defined later in the paper or the *registerUser* command. The return value of the core function depends on the command issued.

In the following we will use the term user to mean a user that runs one or more Harness applications on a host, and we will use the term application to mean a program willing to request and use services provided by the Harness system.

Any application must register via *registerUser* before issuing any command to a Harness VM. Parameters to this command are *userName* and *userPassword*; this call will set a security context object that will be used by the system to check user privileges. When the registration procedure is completed the application can start issuing commands to the Harness system interacting with a local Harness kernel.

A Harness kernel is the interface between any application running on a host and the Harness system. Each host willing to participate in a Harness VM runs one kernel for each VM. The kernel is bootstrapped by the core library during the user registration procedure. A Harness kernel delivers services to user programs and cooperates with other kernels and the VM status server to manage the VM. The status server acts as a repository of a centralized copy of the VM status and as a dispatcher of the events that the kernel entities want to publish to the system (see figure 2 in next page). Each VM has only one status server entity in the sense that all the other entities (kernels) see it as a single monolithic entity with a single access point. Harness VM's use a built-in communication subsystem to distribute system events to the participating active entities. Applications based on message passing may use this substrate or may provide their own communications fabric in the form of a Harness plug-in. In the prototype, native communications use TCP and UDP/IP-multicast.

2.2 Virtual Machine Management: Dynamic Evolution of a Harness VM

In our early prototype of Harness, the scheme we have developed for maintaining the status of a Harness VM is described below. The status of each VM is composed of the following information:

- membership, i.e. the set of participating kernels;
- services, i.e. the set of services that, based on the plug-in modules currently loaded, the VM is able to perform both as a whole and on a per-kernel basis;
- baseline, i.e. the services that new kernels needs to be able to deliver to join the VM and the semantics of these services;

It is important to notice that the VM status is kept completely separated from the internal status of any user application in order to prevent its consistency protocol from constraining users' applications requirements.

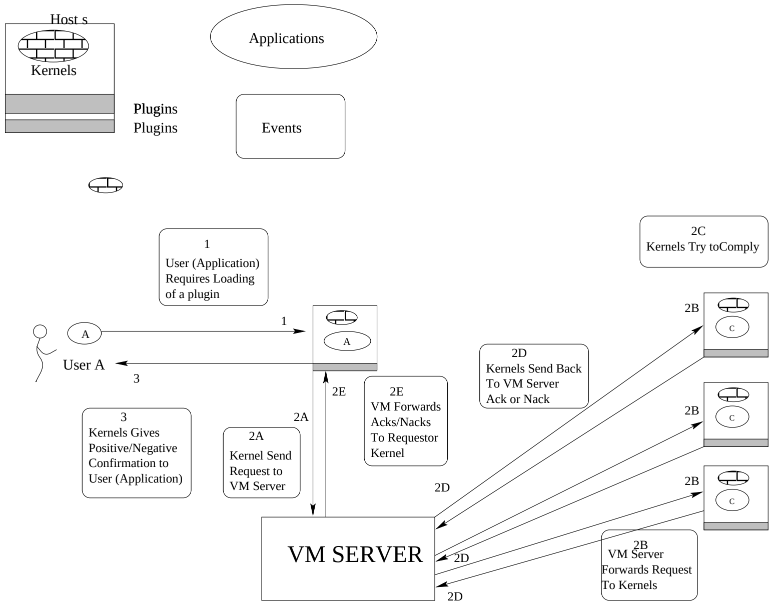

Fig. 2. Event sequence for a distributed plug-in loading

To prevent the status server from being a single point of failure, each VM in the Harness system keeps two copies of its status: one is centralized in the status server and the second collectively maintained among the kernels. This mechanism allows reconstruction of the status of each crashed kernel from the central copy and, in case of status server crash, reconstructing the central copy from the distributed status information held among the kernels.

Each Harness VM is identified by a VM symbolic name. Each VM symbolic name is mapped onto a multicast address by a hashing function. A kernel trying to join a VM multicasts a "join" message on the multicast address obtained by applying the hashing function to the VM symbolic name. The VM server responds by connecting to the inquiring kernel via a reliable unicast channel, checking the kernel baseline and sending back either an acceptance message or a rejection message. All further exchanges take place on the reliable unicast channel. To leave a VM a kernel sends a "leave" message to the VM server. The VM server publishes the event to all the remaining kernels and updates the VM status. Every service that each kernel supports is published by the VM status server to every other kernel in the VM. This mechanism allows each kernel in a Harness VM to define the set of services it is interested in and to keep a selective up-to-date picture of the status of the whole VM. Periodic “I’m alive” messages are used to maintain VM status information; when the server detects a crash, it publishes the event to every other kernel. If and when the kernel rejoins, the VM server gives it the old copy of the status and wait for a new, potentially different, status structure from the rejoined kernel. The new status is checked for compatibility with current VM requirements. A similar procedure is used to detect failure of the VM server and to regenerate a replacement server.

2.3 Services: the User Interface of Harness Kernels

The fundamental service delivered by a Harness kernel is the capability to manipulate the set of services the system is able to perform. The user interface of Harness kernels accepts commands with the following general syntax:

<command> <locator> <targets> <Quality of Service> [additional parameters]

The command field can contain one of the following values:

- **load** to install a plug-in into the system;
- **run** to run a thread to execute plug-in code;
- **unload** to remove an unused plug-in from the system;
- **stop** to terminate the execution of a thread

Services delivered by plug-ins may be shared according to permission attributes set on a per plug-in basis. Users may remove only services not in the *core* category. A *core service* is one that is mandatory for a kernel to interact with the rest of the VM. With the **stop** and **unload** commands a user can reclaim resources from a service that is no longer needed, but the nature of *core services* prevents any user from downgrading a kernel to an inoperable state. However, although it is not possible to change *core services* at run time, they do not represent points of obsolescence in the Harness system. In fact they are implemented as hidden plug-in modules that are loaded into the kernel at bootstrap time and thus easily upgraded. The *core services* of the Harness system form the baseline and must be provided by each kernel that wishes to join a VM. They are:

- the VM server crash recovery procedure;
- the plug-in loader/linker module;
- the core communication subsystem.

Commands must contain the unique locator of the plug-in to be manipulated. The lowest level Harness locator, the one actually accepted by the kernel, is a Uniform Resource Locator (URL). However any user may load at registration time a plug-in module that enhances the resource management capabilities of the kernel by allowing users to adopt Uniform Resource Names (URNs), instead of URLs, as locators. The version of this plugin provided with the basic Harness distribution allows:

- checking for the availability of the plug-in module on multiple local and remote repositories (e.g. a user may simply wish to load the “SparseMatrixSolver” plug-in without specifying the implementation code or its location);
- the resolution of any architecture requirement for impure-Java plug-ins.

However, the level of abstraction at which service negotiation and URN to URL translation will take place, and the actual protocol implementing this procedure, can be enhanced/changed by providing a new resource manager plug-in to kernels.

The target field of a command defines the set of kernels that are required to execute the command. Every non local command is executed using a two phase commit protocol. Each command can be issued with one of the following Quality of Service(QoS): all-or-none and best-effort. A command submitted with a all-or-none QoS succeeds if and only if all of the kernels specified in the target field are able (and willing) to execute it. A command submitted with a best-effort QoS fails if and only if all the kernels specified in the target field are unable (unwilling) to execute it. Both the failure and the success return values include the list of kernel able (willing) to execute the command and the list of the unable (unwilling) ones.

3 Related Works

Metacomputing frameworks have been popular for nearly a decade, when the advent of high end workstations and ubiquitous networking in the late 80's enabled high performance concurrent computing in networked environments. PVM [2] was one of the earliest systems to formulate the metacomputing concept in concrete terms, and explore heterogeneous network computing. PVM however, is inflexible in many respects. For example, multiple DVM merging and splitting is not supported. Two different users cannot interact, cooperate, and share resources and programs within a live PVM machine. PVM uses internet protocols which may preclude the use of specialized network hardware. A “plug-in” paradigm would alleviate all these drawbacks while providing greatly expanded scope and substantial protection against both rigidity and obsolescence.

Legion [3] is a metacomputing system that began as an extension of the Mentat project. Legion can accommodate a heterogeneous mix of geographically distributed high-performance machines and workstations. Legion is an object oriented system where the focus is on providing transparent acess to an enterprise-wide distributed computing framework. As such, it does not attempt to cater to changing needs and it is relatively static in the types of computing models it supports as well as in implementation.

The model of the Millennium system [4] being developed by Microsoft Research is similar to that of Legion's global virtual machine. Logically there is only one global Millennium system composed of distributed objects. However, at any given instance it may be partitioned into many pieces. Partitions may be caused by disconnected or weakly-connected operations. This could be considered similar to the Harness concept of dynamic joining and splitting of DVMs.

Globus [5] is a metacomputing infrastructure which is built upon the “Nexus” [6] communication framework. The Globus system is designed around the concept of a toolkit that consists of the pre-defined modules pertaining to communication, resource allocation, data, etc. Globus even aspires to eventually incorporate Legion as an optional module. This modularity of Globus remains at the metacomputing system level in the sense that modules affect the global composition of the metacomputing substrate.

The above projects envision a much wider-scale view of distributed resources and programming paradigms than Harness. Harness is not being proposed as a world-wide infrastructure, but more in the spirit of PVM, it is a small heterogeneous distributed computing environment that groups of collaborating scientists can use to get their science done. Harness is also seen as a research tool for exploring pluggability and dynamic adaptability within DVMs.

4 Conclusions and Future Work

In this paper we have described our early work on the plug-in mechanism and the dynamic Virtual Machine (VM) management mechanism of the Harness system, an experimental metacomputing system. These mechanisms allow the Harness system to achieve reconfigurability not only in terms of the computers and networks that comprise the VM, but also in the capabilities and the services provided by the VM itself, without compromising the coherency of the programming environment.

Early experience with small example programs show that the system is able:

- to adapt to changing user needs by adding new services via the plug-in mechanism;
- to safely add or remove services to a distributed VM;
- to locate, validate and load locally or remotely stored plug-in modules;
- to cope with network and host failure with a limited overhead;
- to dynamically add and remove hosts to the VM via the dynamic VM management mechanism.

In a future stage of the Harness project we will test these feature on real world applications.

References

1 N. Boden et al., *MYRINET: a Gigabit per Second Local Area Network*, IEEE-Micro, Vol, 15, No. 1, February 1995.

2 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam, *PVM: Parallel Virtual Machine a User’s Guide and Tutorial for Networked Parallel Computing*, MIT Press, Cambridge, MA, 1994.

3 A. Grimshaw, W. Wulf, J. French, A. Weaver and P. Reynolds. *Legion: the next logical step toward a nationwide virtual computer*, Technical Report CS-94-21, University of Virginia, 1994.

4 Microsoft Corporation, *Operating Systems Directions for the Next Millenium*, position paper available at http://www.research.microsoft.com/research/os/Millennium/mgoals.html

5 I. Foster and C. Kesselman, *Globus: a Metacomputing Infrastructure Toolkit*, International Journal of Supercomputing Application, May 1997.

6 I. Foster, C. Kesselman and S. Tuecke, *The Nexus Approach to Integrating Multithreading and Communication*, Journal of Parallel and Distributed Computing, 37:70-82, 1996