HARNESS: A Next Generation Distributed Virtual Machine

Micah Beck [a], Jack J. Dongarra [a,b], Graham E. Fagg [a], G. Al Geist [b], Paul Gray [c], James Kohl [b], Mauro Migliardi [c], Keith Moore [a], Terry Moore [a], Philip Papadopoulous [b], Stephen L. Scott [b] and Vaidy Sunderam [c]

[a] *Department of Computer Science, University of Tennessee, 104 Ayres Hall, Knoxville, Tennessee TN-37996-1301, USA*

[b] *Oak Ridge National Laboratory*

[c] *Emory University*

Abstract

HARNESS (Heterogeneous Adaptable Reconfigurable Networked SystemS) is an experimental metacomputing system[22] built around the services of a highly customizable and reconfigurable distributed virtual machine (DVM). The successful experience of the HARNESS design team with the Parallel Virtual Machine (PVM) project has taught us both the features which make the DVM model so valuable to parallel programmers and the limitations imposed by the PVM design. HARNESS seeks to remove some of those limitations by taking a totally different approach to creating and modifying a DVM.

Keywords: metacomputing, message-passing libraries, distributed applications, distributed virtual machines, PVM

1 Introduction

Virtual machine (VM) terminology, borrowed from PVM[8], refers to the fact that the computing resources on which a system runs can be viewed as a single large distributed memory computer. The *virtual machine* is a software abstraction of a distributed computing platform consisting of a set of cooperating daemon processes. Applications obtain VM services by communicating with daemon processes through system-specific mechanisms encapsulated by a portable API. We define a *Distributed Virtual Machine* (DVM) to be a co-operating set of daemons that together supply the services required to run user programs as if they were on a distributed memory parallel computer. These daemons run on (often heterogeneous) distributed groups of computers connected by one or more networks.

There are three key principles that have guided the design and implementation of existing distributed virtual machine systems, such as PVM:

(I) a simple API, transparent heterogeneity, and dynamic system configuration. The simple API allows messaging, virtual machine control, task control, event notification, event handlers, and a message mailbox all to be accessed and controlled using only about 60 user-level library routines.
(II) Transparent heterogeneity makes it easy to construct programs that interoperate across different machine architectures, networks, programming languages, and operating systems.
(III) Dynamics allow the virtual machine configuration to change and the number of tasks that make up a parallel/distributed computation to grow and shrink under program control. Proponents of PVM have exploited these features and learned to live within the boundaries that the system provides. For example, PVM has always traded off achieving peak performance for heterogeneity and ease of use.

Our next-generation environment will focus on dynamic extensibility while supplying standard MPI[15] and PVM APIs to the user. The ability to adapt and reconfigure the features of the operating environment will enable several significant new classes of applications. The challenge is to implement a reconfigurable substrate that is simultaneously efficient, dynamic, and robust. The initial challanges addressed by the design of the HARNESS DVM are the creation and management of the constituent VM daemons and the core services implemented by the cooperating set of daemons.

1.1 PVM limitations

In PVM, the initial kernel process (or PVM daemon) that is created is the master, and all subsequent daemons are started by the master daemon using the remote shell (rsh) protocol. All system configuration tables are maintained by the master, which must continue running in order for the VM to operate. The communication space of the virtual machine is restricted to the scope of the running set of daemons, therefore, no PVM messages can flow outside the VM, for example, to other VM or outside processes.

The set of services implemented by the PVM kernel is defined by the PVM source code, and the user has only a limited ability to add new services or to change the implementation of standard services. For examples of where such flexibility is needed, consider that the availability of Myrinet interfaces [2] and Illinois Fast Messages [19] has recently led to new models for closely coupled PC clusters. Similarly, multicast protocols and better algorithms for video and audio codecs have led to a number of projects focusing on telepresence over distributed systems. In these instances, the underlying PVM software would need to be changed or re-constructed for stream data, and this would not be trivial.

We see a common theme in all popular distributed computing paradigms, including PVM: each mandates a particular programming style, such as message-passing, and builds a monolithic operating environment into which user programs must fit. MPI-1, for example, prefers an SPMD-style static process model with no job control. This maximizes performance by minimizing dynamics and works very well in static environments. Programs that fit into the MPI system are well served by its speed and rich messaging semantics. PVM, on the other hand, allows programs to dynamically change the number of tasks and add or subtract resources. However, programs in general pay a performance penalty for this flexibility. Even though MPI and PVM provide very useful environments, some programs simply are not able to find the right mix of tools or are paying for unwanted functionality. Here, the monolithic approach breaks down and a more flexible pluggable substrate is needed. This idea is certainly not unique and has been successfully applied in other areas: the Mach operating system [20] is built on the microkernel approach, Linux has "plug-in" modules to extend functionality, and Horus [21] uses a "Lego Block" analogy to build custom network protocols. By extending and generalizing these ideas to parallel/distributed computing, Harness will enable programs programs to customize their operating environment to achieve their own custom performance/functionality tradeoffs.

1.2 The HARNESS approach

HARNESS defines kernel creation in a much more flexible way than existing monolithic systems, viewing a DVM as a set of components connected not by shared heredity to a single master process, but by the use of a shared registry which can be implemented in a distributed, fault tolerant manner. Any particular kernel component thus derives its identity from this robust distributed registry.

Flexibility in service components comes from the fact that the HARNESS daemon supplies DVM services by allowing components which implement those services to be created and installed dynamically. Thus the daemon process, while fixed, imposes only a minimal invocation structure on the DVM.

The HARNESS distributed registry service is used to hold all DVM state. When components are added to the DVM at invocation or runtime, this information is added to the registry. Similarly, the components of two DVMs can be merged *en mass* by merging their respective registries, and some set of components can be split from a DVM by creating a new registry for them and deleting their entries from the old one. These notions of merging and splitting are quite general, but their practicality will be determined by the ease with which a resilient system enabling such dynamic reconfiguration can be built.

When adding a component or set of components to a DVM, the services may be of a kind that already exists within the DVM or they may be entirely new. The addition of a new component can define a new service, or may replace the implementation of a previously defined service by taking its place in the registry. Such extensibility and reconfigurability allows us to consider the new components to be a kind of plug-in, much as operating systems have configurable device drivers and Web browsers have plug-ins for displaying new object types. The addition of a new component may even require applications to load new libraries to make use of them, and we consider such reconfiguration to be a user level component of the plug-in. HARNESS will support a callback feature from kernel components to the user's HARNESS runtime system to enable such reconfiguration to be performed automatically. Thus, a *plug-in* can be defined as a modification to the HARNESS DVM which is composed of a kernel module and/or an application library which seamlessly replace or extend existing system functionality and which can be configured through calls to other system components and changes to the registry.

2 Design Objectives

The HARNESS DVM allows a distributed set of resources to be flexibly managed and exploited for high performance computation. The most important design criteria for HARNESS are:

(1) Flexible management of the components which make up one or more DVMs.
(2) The ability to dynamically modify and extend DVM services (reconfigurable via plug-ins).
(3) The ability of applications (or tools) to collaborate within a DVM.
(4) Management of interactions between multiple DVM users.

HARNESS differs from distributed operating systems in that it is not based on a native kernel that controls the fundamental resources of the constituent computers. Instead, it is built on an operating environment kernel that can be implemented as a process running under some host operating system. The set of user level kernels is said to form a distributed virtual machine (DVM), borrowing the terminology of PVM.

The need for dynamic reconfigurability of the environment ("pluggability")is a challenging design objective that affects the system architecture at every level. At the lowest levels (kernel loading of executable components and an environment of data bindings), we choose very flexible mechanisms for the loading of system components and for the maintenance of system state. These mechanisms make pluggability possible by placing as few limitations as possible on the evolution of the system as it executes. The key features of these low level mechanisms are as follows:

(1) The use of flexible naming schemes for the dynamically changeable sets of system elements.
(2) Minimal set of core functions.
(3) Few restrictions on the types of extensions permitted.

These flexible mechanisms do not define a pluggable system, but merely enable the creation of one. The HARNESS implementation will include default system components that together constitute a complete working system. It will also include additional resource management and communication components to provide flexible functionality not possible in existing systems like PVM. The more difficult challenge is the creation of system components that can make use of this flexible infrastructure. The general problem of dynamic system configuration is a very difficult one that we do not claim to solve. The additional system structures that may have to be created and conventions that must be adopted to achieve overall system reconfigurability remain to be empirically determined by component designers.

3 Architecture

The key architectural features of HARNESS are these:

(1) The kernel is implemented as a set of core functions for loading and running components either locally or remotely. Each component is implemented as a set of calls, processes or threads.
(2) A HARNESS daemon is composed of a kernel and a set of required components. The daemon is an event-driven HARNESS application that responds to requests from a local application or a remote daemon to execute one of its functions. The required components provide message passing, the ability to start processes or threads, the ability to add to the system registry, and the ability to start other kernels.
(3) A HARNESS DVM is composed of a set of cooperating daemons which

together present the basic services of communication, process control, resource management, and fault detection.
(4) A robust registry service is implemented for storing data in a form accessible to any component or application in the DVM.
(5) Mechanisms are provided for the dynamic management of system components, constituting a DVM, through operations on the registry service.

3.1 The HARNESS Kernel

The HARNESS kernel is designed for modularity and extensibility. The kernel itself is a container into which components can be loaded and run. The kernel API is minimal, implementing only a handful of operations on components:

```
VMcontext = registerUser(arg-list)

status = load(VMcontext, component, flags)

status = unload(VMcontext, component)

status = run(VMcontext, component, arg-list)

status = stop(VMcontext, component)

msg = getvminfo(VMcontext, key)
```

where

- VMcontext is a binding of a particular virtual machine ID and a user. This construct allows the kernel to be able to determine authorization and scope of the other operations. This ability becomes even more important when multiple virtual machines (with multiple users) are merged together.
- status returns an error code if the function fails and a handle to the component if it succeeds.
- component is an identifier of a component. Initial implementations use URI's as component names.
- getvminfo returns a message associated with the registry entry tied to key, an example query is to list components currently loaded at a particular kernel.

The most basic service provided by a DVM is an abstract communication method among programs, tools, and virtual machine components. Depending upon the facilities required and the programming environment to be supported in a given HARNESS configuration, different communication components might be used. By default a reliable, ordered delivery of untyped messages to identifiable end-points will be supplied by the communication components within the executing DVM. By rigidly defining inquiry and service interfaces, the HARNESS kernel can determine if requested components meet the requirements of other components in the protocol stack. The research challenge in this regard will be to evolve a methodology for the semantic definition of the interfaces that each plug-in will provide, in a manner that permits interchange and negotiation.

Layered on low level communication (but at a functionally equivalent level in the application interface) are the machine configuration and process control components. For machine configuration, module functionality consists primarily of initialization functions and architecture reconciliation with the rest of the DVM. Our initial HARNESS resource management component will provide a means to add and delete hosts, and to detect host failures within a single DVM. Additional functionality will be developed to add the capability of merging two DVMs based on direct user input, or based on a configuration file that specifies access restrictions.

Process management components will constitute the infrastructure for spawning application task units, and for naming and addressing tasks in the dynamic DVM. Process control modules, similar to ones used in PVM, are under development to provide functions for spawning and terminating groups of tasks across the DVM, using a simple load-balancing algorithms for task placement.

4 The HARNESS Registry Service

The key organizing construct used by kernel components is that of a robust shared registry that maps names to values encoded in a standard format. This registry is used for sharing information between system elements (components and applications) and particularly for system configuration. The registry is implemented by a core component that must be present in at least one kernel. The HARNESS registry is modeled after the PVM 3.4 message mailbox facility[9].

Some uses of the registry in the configuration of the HARNESS system include using it to store:

(1) the list of hosts which constitute a DVM,
(2) the components which must be present in order for a kernel to participate in the current DVM (system level plug-ins),
(3) the list of libraries which applications must load to participate (user level plug-ins).
(4) the list of the dynamic groups of tasks that constitute parallel applica-

tions.

In this sense, the HARNESS registry is similar to the configuration files in the Unix /etc directory, or the Microsoft Windows registry.

The PVM equivalent to the registry are tables kept within the address space of the master PVM daemon, leading to a centralized model which is not robust to the failure of the master. In HARNESS, we require that the registry be robust, meaning that it must be implemented in a distributed, fault tolerant manner as shown in Figure 1. Because of the scope of uses for the registry within HARNESS, we made a design choice regarding the consistency of replicas which specifies that updates to the HARNESS state are seen in the same order everywhere in the system.

The HARNESS registry is an internal tuple space implemented by a distributed set of kernel components. Tasks can use standard routines to encode an arbitrary data item in an architecture-neutral format and then place it into the registry with an associated name. Copies of this data item can be retrieved by any client that knows the name. And if the name is unknown or changing dynamically, then the registry can be queried to find the list of names active in the registry.

The four functions that make up the HARNESS registry API are:

```
index = putinfo( name, itembuf, flag )

recvinfo( name, index, flag )

delinfo( name, index, flag )

getreginfo( pattern, names[], struct info[] )
```

The flag defines the properties of the stored data items, such as who is allowed to delete this item, including control over multiple instances of data items, such as using *putinfo()* to overwrite an existing message instance.

While the tuple space could be used as a distributed-shared memory, similar to the Linda[4] system, the granularity of the message-box implementation is better suited to large grained data storage.

Beyond HARNESS system configuration, there are many potential registry uses, including the following:

(1) A visualization tool spontaneously comes to life and finds out where and how to connect to a large distributed simulation.
(2) A scheduling tool retrieves information left by a resource monitor.

(3) A new team member learns how to connect to an ongoing collaboration.
(4) A debugging tool retrieves a message left by a performance monitor that indicates which of the thousands of tasks is most likely a bottleneck.

Many of these capabilities are directly applicable to the HARNESS environment, and some approach to having persistent messages will be a part of the HARNESS design.

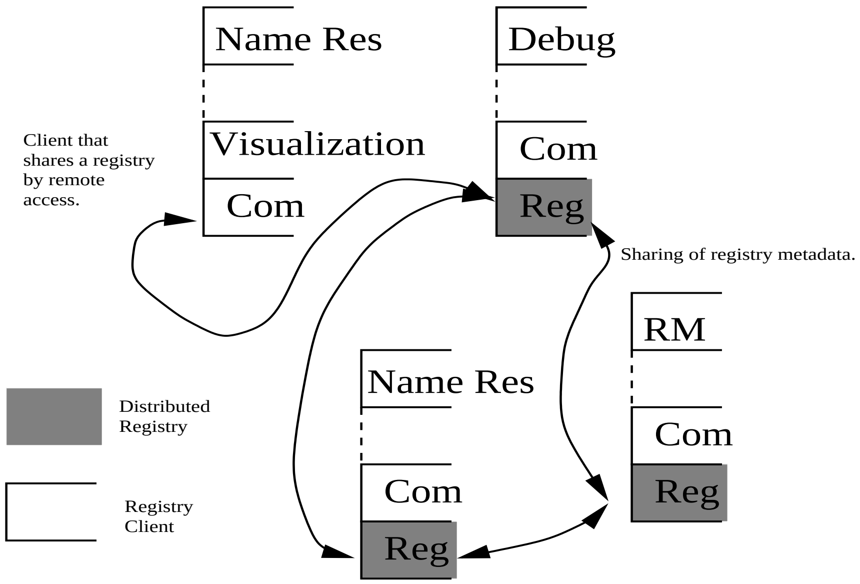

Fig. 1. Distributed Registry and Client Usage.

The addition of communication contexts, message handlers, and message boxes to the parallel virtual machine environment allows developers to take a big leap forward in the capabilities of their distributed applications. HARNESS is a useful tool for the development of much more dynamic, fault tolerant distributed applications, but as illustrated above, it is also a testbed for software and features that will make up the next generation heterogeneous distributed computing environment.

4.1 Merging and Splitting DVM

Important HARNESS design goals include the ability to merge two DVMs to create a single DVM and the ability to split an existing DVM into distinct, functional sub-environments. Understanding a DVM as being defined by its registry, this amounts to judicious manipulation of the registry.

The implementation will need to address issues of communication in the merged DVM and configuration of the combined system:

- Can merging and splitting of the environment occur at any time, or is explicit synchronization with the components and even applications of the constituent machines necessary?

- What view does each resource in a merged environment hold of the Environment? Is this view symmetric amongst resources in distinct groupings?
- What restrictions are placed on utilization of resources in complementary groupings?
- What are the semantics of merging and splitting, and are they uniform across the extent of a DVM? Can merging and splitting be asymmetric?
- Once two DVMs have merged, can one or both retain its original registry, or do both have the new merged registry? How are the registries effected upon splitting of the environments?
- Who in the merged resource pool has the authority to split a conglomerate DVM into sub-environments?
- How can a subsidiary DVM be created, perhaps having the identity of one of the constituent in a merging of DVM (splitting)?
- How will an environment be made aware of other environments?
- After handshaking, what protocols, are to be used so that individual components (computational resources, processes and the like) may have uninhibited communication with any or all components of the complement.
- How will the combined environment be reconfigured to reestablish all necessary system properties? If some module is required in every kernel of one environment, will it automatically be loaded on every kernel of the other? Might new modules be needed to bridge existing elements of the two environments?

For insight on dealing with these issues, approaches taken by other systems such as PVM, Legion[11], Globus[7], and IceT[10] provide initial prototypes for this merging of environments.

Once the two systems are joined together in communication and sufficiently configured for functionality, there is an additional and unique facility of HARNESS, viz. HARNESS will provide seamless process creation on any host in the combined resource pool. Here, the issue is how a process which is part of one virtual environment is to be ferried across distributed environmental boundaries for execution on a possibly foriegn operating system or architecture. There have been some preliminary results at implementing this cross-environment functionality in IceT. IceT, in an early prototype, utilizes aspects of portability found in the Java programming language to port both Java-based and C/Fortran processes across system boundaries. However, the applicability of IceT's process location, process creation, and security implementations relative to the more broadly-defined goals and objectives of HARNESS have yet to be determined.

Merging of environments will be prefaced by the need to gain information on where and how to handshake with outside environments. For storing such information, the Resource Catalog (RC) will be used (see section 6). An environment wishing to attach to another would query the RC server for information on listening ports, communication protocols, module configuration, and security restrictions. With this information, the environments may initiate contact, share state information, and update their registries and the RC server to reflect the new state of the system. The state information which is passed between environments includes information about the components of the computational resources enrolled in the respective environments. Information about the computational components is in tableau form, with provides information for each resource, such as "operating system," "architecture," "modules loaded," "modules available," "host name", "listening ports," and "accessibility levels."

Splitting a DVM into distinct, yet functional, sub-environments is much more of a challenge. As such, splitting functionality will be incorporated into the distributed environment vis-a-vis a "splitting plug-in". This splitting plug-in defines which of the entities will be allowed to secede from the environment, which entities will have the authority to sub-divide resources, what to do with messages intended for resources recently split apart, etc. For example, one might configure the splitting module to hold messages sent to split-off resources in a message box which would be passed along once the sub-DVMs rejoin (re-merge), or to disallow secession of groups involving local networked resources and nonlocal resources.

5 Configuring HARNESS: Communication

As distributed computing has developed, it has become clear that no one monolithic system can efficiently handle all the desired communication styles. Extensibility of the core system is essential to achieve critical performance and provides a practical method to manage multiple communication styles. Because messaging is extremely important to system performance and evaluation, the lowest layers must be able to be swapped out for different scenarios. For example, send/receive message passing is quite different from one-sided communication semantics. Low-level performance can be significantly affected if support for both is automatically installed even though not needed by an application. The inefficiency comes from the fact that incoming messages need to be checked among different communication methods to determine the correct handling sequence. If a particular message style (e.g. a put) is never used, then eliminating this as a checked-for style can produce a reduction in overhead. On MPPs, for example, it is unnecessary to fragment messages or provide reliable transmission because it is usually guaranteed by the message system. On the other hand, communicating over a WAN requires fragmentation, timeouts/retries, and connection failure monitoring. A user should be able to write a distributed application and have the runtime system select which method(s) are needed for the particular run. The key to success will be to design plug-in communication stacks (similar to those found in Horus [21]) that can be traversed quickly and efficiently. To get optimum performance, it may require the user to use strongly-typed messaging like MPI. However, runtime configurability can still give significant advantages without requiring users to dramatically change code. For example, one may desire to encrypt certain communication links only if the virtual machine spans several domains. Runtime configurability will allow an encrypted link to be installed without user code modification. The next generation DVM will have to strike a better balance among performance (or the ability to optimize performance), extensibility, and interoperability. Due to the large body of research on communication methods, this lowest level of pluggability is probably the most straightforward goal to achieve.

6 Name Resolution

While HARNESS focuses on the management of distributed resources within a DVM, in today's computing environment it is also necessary to deal with network resources outside of the DVM. PVM did not support any access to outside resources, leaving each application process to implement such access independently. HARNESS is more general than PVM, allowing for the communication with and assimilation of resources outside the DVM.

The goal of the HARNESS system is to provide a scalable and robust name resolution service such as the resolution scheme implemented by the Resource Catalog [18]. The Resource Catalog is a simple, highly available, and very scalable distributed resolution service.

By *resolution service*, we mean a service for mapping a resource name onto a set of attributes or characteristics of a resource, which are sometimes called metadata. A resolution service differs from a directory service such as X.500 [12], in that a resolution service maps a name onto its associated attributes, while a directory service is intended to allow searching of the attributes themselves to identify matching resources. Common resolution services include the Domain Name System (DNS)[17] used in the Internet and the Network Information Services (NIS) used on UNIX systems. In contrast to these, the Resource Catalog was designed to be simple, flexible, efficient, reliable, fault-tolerant, secure, and very scalable.

The Resource Catalog is distributed in that the set of resource characteristics are maintained by an arbitrary number of servers on an arbitrary number of network hosts. Each server contains the resource metadata for each of the resources in a well-defined subset of the resource name space. The metadata for any resource may be replicated across several servers to improve scalability and availability. Updates to a resource's metadata may be made to any of the servers that maintains that resource's metadata, using a discipline that ensures that any client will see all of the updates from any one source in the same order. Multiple parties may update the metadata for a single resource (given the proper permission and security credentials). Each party's updates to a particualar resource characteristic are kept separate and returned together in the same response; no party's updates may override another's.

The resource names used by the Resource Catalog are in the form of Uniform Resource Identifiers (URIs). URIs are a slight generalization of Uniform Resource Locators (URLs), and the set of URLs is a subset of URIs. URLs are generally understood to refer to a particular location of a resource (e.g. a specific file on a specific host), less can be assumed about a URI: it is merely the name of a resource. Rather than parsing a URL to determine a particular protocol, host, port, and filename; an application submits a URI to the Resource Catalog to determine information about that resource. Various kinds of information may be returned, for example: the current location(s) of the resource, the owner of the resource, the permissions associated with the resource, the public key to be used when securely communicating with the resource, the date that the resource was last modified.

The metadata for a resource consists of a set of assertions. Each assertion is a characteristic consisting primarily of a name, which is a NUL-terminated string, and a value, which is an opaque string of octets. A type field is also provided to aid applications that might wish to display or otherwise interpret the data. Each assertion also contains the identify of the party that made the particular assertion about the resource, the date and time at which the assertion was made, the serial number of that assertion (i.e. the number of times that that party had changed the value associated with the assertion). Finally, each assertion contains a time-to-live field and an expiration date that can be used to determine the amount of time that metadata is cached.

HARNESS uses the Resource Catalog to store information about DVMs (including the set of hosts in the DVM, and the means by which other DVMs, or external processes, can communicate with the DVM), individual hosts (including host characteristics, public keys, and other information used during negotiation of network connections with that host), mobile processes (including their current location and contact information), and plug-ins (their current locations, host requirements, and digital signatures).

7 Results

The HARNESS system is based on the ability to perform three operations.

(1) Plug-in new features or functionality into the kernel of a DVM
(2) Have two independently started applications discover each other and co-operate
(3) Merge two DVM together

We have demostrated each of these capabilities in separate DVM prototypes and our effort in HARNESS is to incorporate all these capabilities into a single compact system.

Version 3.4 of PVM has three specific plug-in interfaces–one for task scheduling, one for task creation, and one for adding hosts to a virtual machine. These plug-ins allow these three capabilities to be dynamically replaced with user written modules during runtime. Several groups both industrial and academic use these plug-in interfaces to intergrate their own software into the virtual machine environment supplied by PVM. The goal of HARNESS is to now generalize this result and create an environment where nearly every feature in a DVM can be replaced with a user supplied version and where new features previously not available in the DVM can be added. We have a working prototpe of this generalized plug-in interface running at Emory University.

We have spent the last two years working on a remote computational steering environment called Cumulvs [14]. In Cumulvs any number of independent "viewer" applications can spontaneously come to life, discover if there are any distributed applications running on the DVM that are Cumulvs enabled, and attach to their applications. Once attached these "viewers" can extract data for viewing, or change physical parameters inside the running application. These capabilities, which fall under the second class of operations needed in HARNESS, are made possible by the addition of message-box features in PVM 3.4. It is these same features we plan to leverage in the HARNESS project. One major change needed for HARNESS is to make this registry both robust, and able to guarantee a consistent order of updates across the distributed copies.

IceT [10], developed to be the computation framework for the Collaborative Computing Framework project at Emory University, has demonstrated the capability to merge two DVM. We plan to leverage both the experience and software technology developed for IceT in the HARNESS system. The IceT system serves as a prototype and proof of concept that multiple DVM can be merged together in temporary cooperative environments. The next challenge for HARNESS development in this area is how to merge the multiple distributed registries. Another challenge in HARNESS is how to integrate this capability, which brings multiple users and administrative domains, with the other plug-in features in HARNESS.

8 Related Work

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 was one of the earliest systems to formulate the metacomputing concept in concrete virtual machine and programming-environment terms, and explore heterogeneous network computing. PVM is based on the notion of a dynamic, user-specified host pool, over which software emulates a generalized concurrent computing resource. Dynamic process management coupled with strongly typed heterogeneous message passing in PVM provides an effective environment for distributed memory parallel programs. PVM however, is inflexible in many respects that can be constraining to the next generation of metacomputing and collaborative applications.

Legion[11] 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 access 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 Globe project[13] is related to Legion in that it deals with distributed objects that are used to build large-scale distributed systems. Local object representatives hide details like replication and mobility. Local objects have a standard internal structure that makes it easier to reuse code components. One of Globe's major features is a hierarchical distributed location service that adapts dynamically to different usage patterns.

The model of the Millennium system [3] 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[7] 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.

SNIPE[5] is to metacomputing systems as Unix is to operating systems. It is a distributed systems testbed that provides much of the functionality of systems like PVM without the rigid definition of a virtual machine. It provides process control like PVM based on daemons and resource managers. Communication based on both socket and message based abstractions are built on a layered substrate similiar to that of Nexus. Naming, registry and resource information storage is built using a modified version of RCDS that provides global naming based on URIs. Many of the lessons learnt building SNIPE will go towards the naming, registry, resource discovery and multi-path communications sections of Harness research.

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.

9 Conclusions

The account we have given of HARNESS both motivates the need for a next generation DVM model and presents primitive mechanisms that address key requirements of this new model. HARNESS' modular kernel architecture supports a level of flexibility in the set of system components that is not available under monolithic operating environments such as PVM and MPI. HARNESS' system registry allows distributed control of the system configuration in order to enable the dynamic addition of new components and libraries, as well as the merging and splitting of distinct virtual machines. These flexible mechanisms do not fully define how dynamic reconfiguration will proceed, but merely make such a reconfigurable DVM possible. The more difficult challenge is the creation of system components that can make use of this flexible infrastructure, and that is the enterprise in which we are now engaged.

References

[1] M. Baker, G. Fox, and H. Yau. *Cluster Computing Review.* Northeast Parallel Architectures Center, Syracuse University, November 1995, New York, http://www.npar.syr.edu/techreports/index.html.

[2] N. J. Boden at al., "MYRINET: A gigabit per second local area network", *IEEE-Micro*, Vol. 15, No.1, February 1995, pp.29-36.

[3] William J. Bolosky, Richard P. Draves, Robert P. Fitzgerald, Christopher W. Fraser, Michael B. Jones, Todd B. Knoblock, and Rick Rashid. "Operating System Directions for the Next Millennium". *Position paper of MicroSoft Research*, January 1997.

[4] Nicholas Carriero, David Gelernter, and Jerrold Leichter. Distributed Data Structures in Linda, *Thirteenth ACM Symposium on Principles of Programming Languages Conf.*, St. Petersburg, Florida, Jan. 1986, pp. 236-242.

[5] Graham E. Fagg, Keith Moore, Jack J. Dongarra and Al Geist, *Scalable Networked Information Processing Environment (SNIPE)*, Proceeding of SuperComputing 97, San Jose, CA., November 1997.

[6] I. Foster, C. Kesselman and S. Tuecke. "The Nexus approach to integrating Multithreading and Communication", *Parallel and Distributed Computing*, Vol. 37, pp 70-82, 1996.

[7] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, Vol. 11(2), summer 1997.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Liang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual machine – A User's Guide and Tutorial for Network Parallel Computing,* MIT Press, 1994.

[9] G. A. Geist II, J. A. Kohl, R. Manchek, P. M. Papadopoulos, "New Features of PVM 3.4, " *1995 EuroPVM User's Group Meeting*, Lyon, France, Pub. Hermes, pp. 1-9, September 1995.

[10] P. Gray and V. Sunderam, IceT: Distributed Computing and Java, *Concurrency, Practice and Experience*, ed. Geoffrey C. Fox, Vol. 9 (11), pp 1161-1168, Nov. 1997.

[11] A. S. Grimshaw and W. A. Wulf. The Legion vision of a worldwide virtual computer. *CACM*, Vol. 40(1), pp39-45, January 1997.

[12] S. Heker, J. Reynolds, and C. Weider. Technical overview of directory services using the x.500 protocol. *RFC 1309*, FY14, IETF, 03/12 92.

[13] Philip Homburg, Maarten van Steen, and Andrew S. Tanenbaum. "An Architecture for a Wide Area Distributed System." *In Proceedings of the Seventh ACM SIGOPS European Workshop*, Connemara, Ireland, September 1996.

[14] J. Kohl, P. Papadopoulos, and A. Geist. "CUMULVS: Collaborative Infrastructure for Developing Distributed Simulations.", *In Proc. Eigth SIAM Conf. on Par. Proc. and Sci. Comp.*, Minneapolis, MN, March 1997.

[15] Message Passing Interface Forum, MPI: A message–passing interface standard, *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[16] Message Passing Interface Forum, *MPI-2 Extensions to the Message-Passing Interface*, University of Tennessee, Knoxvile, Tennessee, 1997.

[17] P. Mockapetris, "Domain Names - Concepts and Facilities.", *RFC 1034*, Internet Network Information Center, 1987.

[18] K. Moore, S. Browne, J. Cox and J. Gettler, *The Resource Cataloging and Distribution System*, Technical report, Computer Science Department, University of Tennessee, December 1996.

[19] Pakin, Karamcheti and Chien. "Fast Messages (FM): Efficient, Portable Communication for Workstation Clusters and Massively-Parallel Processors." *IEEE Concurrency*, Vol. 5(2), 1997, pp. 60-73.

[20] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr and Richard Sanzi. "Mach: A Foundation for Open Systems", *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*, September 1989.

[21] Robbert van Renesse, Kenneth P. Birman and Silvano Maffeis, Horus, a flexible Group Communication System, *Communications of the ACM*, April 1996.

[22] L. Smarr and C.E. Catlett. *Metacomputing. Communications of the ACM*, Vol. 35(6), 1992, pp. 45–52.