

University of Windsor

Scholarship at UWindor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

2005

Flexible multi-layer virtual machine design for virtual laboratory in distributed systems and grids.

Dohan Kim
University of Windsor

Follow this and additional works at: <https://scholar.uwindsor.ca/etd>

Recommended Citation

Kim, Dohan, "Flexible multi-layer virtual machine design for virtual laboratory in distributed systems and grids." (2005). *Electronic Theses and Dissertations*. 2119.
<https://scholar.uwindsor.ca/etd/2119>

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

**Flexible multi-layer virtual machine design
for virtual laboratory
in distributed systems and grids**

by

Dohan Kim

A Thesis

**Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor**

Windsor, Ontario, Canada 2005

©2005, Dohan Kim



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-494-09786-8

Our file Notre référence

ISBN: 0-494-09786-8

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

ABSTRACT

Virtual laboratories recently began to provide environments for modelling and simulation of scientific problems, such as DNA sequencing, aerodynamic analysis of aircraft design, and global weather prediction. Although the performance of commodity computers and networks has been increased significantly, these scientific problems have not been handled in an efficient manner within a reasonable time frame in a single administrative domain due to their size and complexity. Computational grids provide the means by which geographically dispersed administrative domains share their resources in a coordinated way, allowing virtual laboratory users to have illusions that they are accessing a large virtual computer.

We propose a flexible Multi-layer Virtual Machine (MVM) design intended to improve efficiencies in distributed and grid computing and to overcome the known current problems that exist within traditional virtual machine architectures and those used in distributed and grid systems. This thesis presents a novel approach to building a virtual laboratory to support e-science by adapting MVMs within the distributed systems and grids, thereby providing enhanced flexibility and reconfigurability by raising the level of abstraction. The MVM consists of three layers. They are OS-level VM, queue VMs, and components VMs. The group of MVMs provides the virtualized resources, virtualized networks, and reconfigurable components layer for virtual laboratories. We demonstrate how our reconfigurable virtual machine can allow software designers and developers to reuse parallel communication patterns. In our framework, the virtual machines can be created “on-demand” and their applications can be distributed at the source-code level, compiled and instantiated in runtime.

DEDICATION

**To my grandmother for her endless love,
To my parents for their unwavering support and love**

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. Kent, who gave me the opportunity to work on this Grid research. This thesis work had been a great learning experience for me, and I would like to thank him for his advice and guidance on this research.

I would like to express my special thanks to my committee members Dr. Schlesinger and Dr. Aggarwal. I would like to thank Dr. Aggarwal for his professional advice on this research. I feel really fortunate that he is my committee member. I would like to express my gratitude to him for his valuable suggestions. I am also highly grateful to Dr. Ngom who agreed to chair the committee.

I would like to thank my colleagues at University of Windsor for their help and advice during this research.

Finally, I would like to thank all my family and friends for all their support.

TABLE OF CONTENTS

ABSTRACT.....	iii
DEDICATION.....	iv
ACKNOWLEDGEMENTS.....	v
LIST OF TABLES.....	viii
LIST OF FIGURES.....	ix
1 INTRODUCTION.....	1
2 BACKGROUND ANALYSIS AND RELATED WORKS.....	4
2.1 Virtual Laboratory.....	4
2.2 Distributed Systems.....	4
2.3 Grids.....	4
2.4 Categories of Virtual Machines.....	5
2.4.1 OS-level VMs and Virtual Machine Monitor (VMM).....	6
2.4.2 Virtual OS.....	8
2.4.3 Language-level VMs.....	8
2.4.4 Queue-based Virtual Machines.....	9
2.5 Process / VM Migration.....	9
2.6 Resource Partitioning.....	10
2.7 Hardware Virtualization.....	11
2.7.1 CPU Virtualization.....	11
2.7.2 Memory and I/O Virtualization.....	11
2.7.3 Full Virtualization.....	12
2.7.4 Paravirtualization.....	13
2.8 Network Virtualization.....	14
2.8.1 Programmable Network.....	14
2.8.2 Network Components Virtualization.....	15
2.9 Model Driven Software Engineering.....	16
2.9.1 Model and Metamodel.....	16
2.9.2 Model Driven Architecture.....	17
2.9.3 Software Component Modeling.....	18
2.10 Related works.....	19
2.10.1 Grid Computing on OS-level Virtual Machines.....	19
2.10.2 Virtual OS based Distributed Systems.....	20
2.10.3 VM TestBed.....	21
3 A PROPOSED VIRTUAL MACHINE (MVM).....	23
3.1 Building a virtual computer using MVMs.....	24
3.2 OS-level VM layer.....	26
3.3 Queue VM Layer.....	28

3.4	Components VM Layer.....	29
3.5	The type (b) and type (c) MVM.....	30
3.6	Application Model.....	32
4	A PROPOSED FRAMEWORK FOR VIRTUAL LABORATORY.....	33
4.1	Virtualized Resources.....	34
4.1.1	Virtual Back-Ends.....	34
4.1.2	Virtual Front-Ends.....	37
4.1.3	Virtual Clusters.....	39
4.2	Virtualized Networks.....	41
4.2.1	Connectivity component.....	41
4.2.2	Partitioning and mapping procedure.....	44
4.3	Policy-based Reconfigurable Components.....	49
5	IMPLEMENTATION AND PERFORMANCE ANALYSIS.....	53
5.1	Overview of MVM toolkit.....	53
5.2	Implementation and specification of the MVM toolkit.....	57
5.2.1	Implementation of the MVM toolkit.....	57
5.2.2	MVM toolkit v.0.1.0 protocol description.....	60
5.3	Experimental results and analysis.....	63
6	CONCLUSION AND FUTURE WORK.....	68
6.1	Conclusion.....	68
6.2	Future work.....	69
	BIBLIOGRAPHY.....	70
	APPENDIX: Interoperability and porting issue with MVM toolkit.....	79
	VITA AUCTORIS.....	82

LIST OF TABLES

Table 1	Four Modeling layers of the OMG [OMG01].....	16
Table 2	A usage of virtual topology in a MPI program.....	41
Table 3	The updated super peer list in the bootstrap node.....	45
Table 4	MVM toolkit v.0.1.0 Command Messages.....	60
Table 5	MVM toolkit v.0.1.0 Response Messages (SUCCESS).....	61
Table 6	MVM toolkit v.0.1.0 Response Messages (FAILURE).....	62

LIST OF FIGURES

Figure 1	(a) Instruction Set Architecture (ISA) Interface [Smit01, page: 7].....	6
	(b) Application Binary Interface (ABI) Interface [Smit01, page: 7].....	6
Figure 2	Virtual switch and steps of port creation [Jian03, page : 5].....	15
Figure 3	MDA Process [Klas04].....	17
Figure 4	Architecture for a VM-based Grid Service [Figu03, page: 12].....	19
Figure 5	Virtual Internetworking on Overlay Infrastructure [Jian03, page: 2].....	22
Figure 6	Multi-layer Virtual Machine Architecture (type (a)).....	23
Figure 7	Virtual Parallel Computer using MVMs.....	25
Figure 8	Multi-layer Virtual Machine Architecture (type (b)).....	30
Figure 9	Multi-layer Virtual Machine Architecture (type (c)).....	31
Figure 10	Job and its header for MVM application.....	32
Figure 11	Proposed Virtual Laboratory Framework.....	34
Figure 12	Virtual Front-End and Virtual Cluster.....	38
Figure 13	Simple Virtual Topology.....	42
Figure 14	Connectivity Component.....	43
Figure 15	Policy-integration for multi-domain environments [Josh04, pp: 50].....	49
Figure 16	Initial phase of MVM (No MVM process loaded).....	55
Figure 17	MVM toolkit starts its operation by SOAP invocation.....	55
Figure 18	Spawns VM threads by on-demand method.....	55
Figure 19	Awakes other nodes in a job group by SOAP invocation	56
Figure 20	UML Diagram of core MVM toolkit modules.....	58
Figure 21	Running MVM toolkit on Globus 3.2.1.....	64
Figure 22	Elapsed time for 3 and 5 MVM nodes for the “random walk” program.....	65
Figure 23	Elapsed time for random walk program (MPICH2 & MVM).....	65
Figure 24	Latency and throughput with different message sizes.....	66

1 INTRODUCTION

In recent decades, virtual laboratories have provided an environment to test various research models, in the modelling phase, for cost and time saving [Pren00]. A virtual laboratory can be dynamically organized, whereby its topology is adapted on-demand by research communities. Our goal is to create a virtual laboratory framework, using multi-layer virtual machines, in which virtual machines are scalable, reconfigurable, and flexible. Our design goals are as follows:

(1) Reconfigurability: The group of virtual machines is able to be reconfigurable for particular uses. Reconfigurable virtual machines allow virtual laboratory users to test various research models for reduced cost and time.

(2) Flexibility: The proposed framework will consist of a wide variety of machines with different architectures; virtual laboratory users do not need to be aware of such details. The proposed framework should not define the internal structure of any of the service components including data and resource types. It is up to the virtual laboratory users or experimenters to (re)configure test models.

(3) Decentralized Maintenance: The proposed framework is organized by communities in the virtual laboratory and each community should be organized in a decentralized way. For easy maintenance, each community is organized as a virtual cluster for the test model experiment, where virtual machines are reconfigurable for each usage case.

(4) Dynamicity: The proposed framework should also allow virtual machines to be created on-demand and dynamically instantiated through real or virtual networks while providing a simplified and logical view of the underlying systems.

(5) Simplicity: The proposed framework should provide virtual laboratory users with an efficient access point or interface to a group of host systems, as if they were using a single system.

(6) Fault/Attack Isolation: Each host system is capable of running multiple test models and if one test model suffers from faults or attacks, side effects of the system should be minimized.

However, these objectives cannot be perfectly met with current system designs, as demonstrated by the following example usage cases.

(1) Developers or experimentalists create complex configurations for parallel computations on current grid middleware. Then they experiment with basically the same programs but different configurations on different architectures. In current system designs, grid middleware is not fully integrated with a wide variety of dynamic configuration model, therefore, we should use the configuration model every deploy-time rather than run time.

(2) Developers or experimentalists create high performance pipelined applications on current virtual machines with multiple stages of connections. Then they experiment with basically the same programs but different stages of connections. In current system designs, virtual machines are not integrated with dynamic connection model for each stage, therefore, either programming efforts or different deploy-time configurations are required to specify the connection model for each stage.

(3) Developers or experimentalists create a virtual topology for parallel applications. Then they experiment with basically the same programs, but with different virtual topologies using the same or different number of nodes. In current parallel system designs, for instance, MPI, we have to rewrite and recompile the program, to test another virtual topology model in the same program.

(4) Developers or experimentalists send jobs to the resource broker that performs job discovery, scheduling, and management on a group of resources. Then they want to change the behaviour of the resource broker, in that some nomadic resources are added to the existing resources. In current system designs, resource brokers are static entities and cannot be dynamically adapted for a wide variety of resource usage models.

The hypothesis of the thesis is as follows:

“We assert that the reconfigurable and layered virtual machine can overcome the known above problems and provide the enhanced efficiency and reconfigurability for virtual laboratory in distributed and grid systems in comparison with currently established methods and techniques.”

This paper presents the new virtual machine architecture for a virtual laboratory framework that meets the goals described above. The new architecture allows a set of virtual machines to be connected together for a larger virtual computer, while virtual machines are reconfigurable for each resource, job, and connectivity scenario. This multi-layer, flexible virtual machine architecture should allow virtual laboratory framework to gracefully scale to a large number of nodes, and allow us to reconfigure the virtual machine itself at run time rather than deploy-time. We divide a group of virtual machines by the general processing element nodes and control processing element nodes. The control processing element nodes act as traditional resource brokers, allowing us to reconfigure and adapt for a wide variety of resource usage models.

The rest of this paper is organized as follows. Chapter 2 presents a background analysis for the thesis. Chapter 3 describes the design of the Multi-layer Virtual Machine (MVM) for the virtual laboratory, and its three layers. We discuss the functionality of each layer, and the design consideration for each layer. Chapter 4 discusses the design of the virtual laboratory framework by using MVMs. It also includes the reconfigurable policy and connectivity components. Chapter 5 presents an implementation and observations from a virtual laboratory using MVMs. Finally, Chapter 6 presents concluding remarks.

2 BACKGROUND ANALYSIS AND RELATED WORKS

2.1 Virtual Laboratory

According to Preney et al [Pren00], a virtual laboratory is a Turing compliant modeling environment where discussions, data exchange and experiments all take place, providing a modeling system environment to model arbitrary models. The ideal virtual laboratory, defined by Preney et al [Pren00], allows collaborating researchers to integrate and utilize the following features within a virtual administrative domain : (a) create, publish, execute, modify, destroy and test their research models; (b) query databases of existing knowledge and models; (c) select machines, algorithms, and data structures in order to generate and deploy software codes necessary to publish or execute a given research model automatically with minimal researcher assistance; (d) enforce the security, copyright, administrative, costing, and information policies of the involved parties. Building an ideal virtual laboratory, as specified above is still a work in progress, and theoretical limits (i.e. Godel's incompleteness theorem) exposed by the modeling itself still need to be overcome [Pren00].

2.2 Distributed Systems

Tanenbaum [Tane95] defines a distributed system as, "a collection of independent computers that appear to the users of the system as a single computer", and clarified two aspects with this definition. The first clarification concerns hardware, where the distributed system is composed of a collection of independent computers, and the second clarification concerns software; the user can think of the distributed system as one single computer [Tane95].

2.3 Grids

According to Foster [Fost01], distributed-computing technology should evolve to accommodate the range of resource types (including the flexibility and control in sharing relationships) to establish dynamic collections of individuals, institutions, and resources;

a "grid problem" is defined as "coordinated resource-sharing and problem-solving in dynamic, multi-institutional virtual organizations". These dynamic collections are called "Virtual Organizations" or VOs [Fost01]. Research and development efforts to build scalable VOs have been conducted in grid communities, but a challenging problem still remains: the achievement of seamless, dynamic, cross-organizational VO sharing. "Open grid services architecture", defined standard mechanisms for the creating, discovering, and naming of "grid service instances", while "service" is defined as a network-enabled entity providing some capability [Fost02]. The benefits of service-oriented grid architectures in combination with web services technologies have been explained by the former's ability to make use of advantageous characteristics (such as service description, discovery, and binding of service descriptions to network protocols) of the latter, with these "virtualized" services allowing consistent resource access across multiple heterogeneous environments by location transparency, and also multiple logical resource instances to be mapped onto the same physical resource [Fost02].

2.4 Categories of Virtual Machines

According to Goldberg [Gold74], a "virtual machine" (VM) is defined as, "an isolated duplicate of a real existing computer system, in which statistically-dominant subset of the virtual processor's instructions are being executed on host processor in a native way". Virtual machines have been studied since the late 1960s and are experiencing resurgence in commercial and research areas. OS-level virtual machines, in addition to virtual OSes, language-level virtual machines and queue-based virtual machines can be deployed in distributed systems and grid environments.

A typical computer system has three components; these are hardware, operating system, and application programs. There are two key interfaces in a typical computer system; respectively, these are called ISA and ABI (See Figure 1 below).

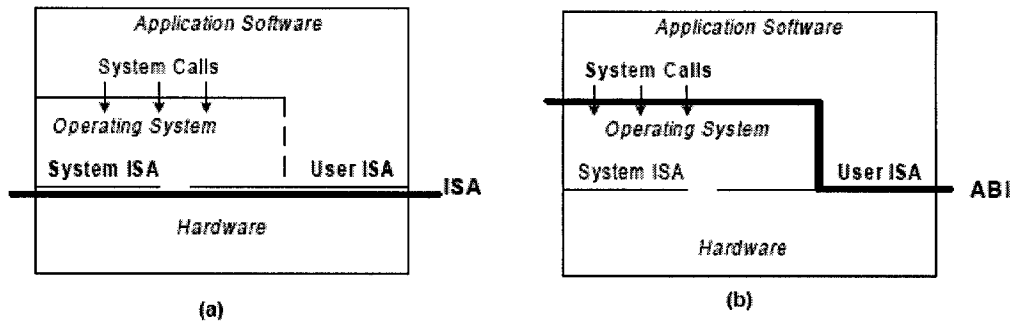


Figure 1 [Smit01, page: 7]

(a) Instruction Set Architecture (ISA) Interface

(b) Application Binary Interface (ABI) Interface

In Figure 1 (a), the ISA interface consists of both the user ISA (non-privileged instruction set architecture) and the system ISA (privileged instruction set architecture). The user ISA is available to both the operating system and application software, whereas the system ISA is only available to the operating system; only privileged operations can be permitted to manipulate the processor, memory and I/O directly [Smit01]. As shown in Figure 1 (b), the main components of ABI interface are the user ISA and “system calls”. Between application programs and an operating system, the “system calls” interface is provided to manage and protect hardware resources from unauthorized accesses [Smit01]. Two major types of virtual machines are ISA VMs and ABI VMs; ISA VMs manipulate and support both the user ISA and the system ISA, whereas ABI VMs manipulate and support both user ISA and “system calls” [Smit01]. OS-level VMs [Wald02] are in the former category but language-level VMs (e.g. Java Virtual Machine) and virtual OSe [Jian03a], [Dike01] are in the latter.

2.4.1 OS-level VMs and Virtual Machine Monitor (VMM)

OS-level VMs (i.e. classic VMs) are categorized as ISA VMs, with the same ISA execution environments of the entire operating systems [Figu03]. “Virtual machine monitor” [Shri76] is involved in the OS-level VMs between hardware and guest operating systems (VMs). The original purpose of OS-level virtual machines is to

multiplex expensive hardware resources by virtualization of the hardware interface; this virtualization of the hardware layer, called a virtual-machine monitor (VMM), allows multiple, concurrent guest OSes to be hosted on the same hardware platform [Crea81]. “Disco” projects [Bugn97] and commercial “VMware” [Suge01] adapt VMM approaches; “Disco” projects [Bugn97] adapts VMM to run multiple commodity operating systems on a scalable multiprocessor, whereas “VMware” [Suge01] adopts VMM to run several guest OSes on intel-based PC platform. VMM emulates underlying hardware resources for guest OSes, with guest OSes then operating as though they are real hardware; in this case, all I/O and privileged instructions must be trapped by VMM, with the VMM also ensuring correct scheduling of CPU time slice amongst guest OSes [Crea81]. Other purposes, such as providing isolation and security, were considered for these guest operating systems on VMM; specifically, because of inherent VMM characteristics, it protects other guest operating systems if a malicious user compromises one underlying guest operating system, compelling him or her to break one more level (i.e., VMM) in order to compromise an entire system [Figu03].

OS-level VMs also provide security monitoring services such as intrusion prevention/detection [Garf03] and secure logging [King02] systems. OS-level VMs are useful for intrusion prevention and detection systems; because running doubtful events on real systems (to test attacks) might compromise the system, a better approach is to clone the real system by OS-level VMs in order to test suspicious events [Chen01]. Secure logging systems through OS-level VMs [King02] has been demonstrated. A major shortcoming of current intrusion logging systems can be the fact that an attacker, after he/she takes control over the system, can easily alter the logging system so it cannot be trusted; moving logging software out of the operating system and into the virtual-machine monitor can help to replay the operating system’s execution before, during, and after any potential attacker compromises the system [King02].

Another advantage of OS-level VMs allows running unmodified-legacy applications to migrate and operate seamlessly without residual dependencies, because the entire virtual machine state can be encapsulated and migrated between VMMs [Osma02], [Sapu02].

2.4.2 Virtual OS

Virtual OS technology [Dike01], [Jian04] is similar to OS-level VM technology in that it allows multiple guest operating systems (virtual OSes) to be hosted on the same hardware platform with fault/attack isolation [Jian04] but its basic difference lies in the fact there is no hardware emulation layer in the virtual OS technology [Dike01]. The virtual OS kernel and its processes run as processes on the host kernel; as a result, a user space virtual machine, using simulated hardware, can be run by the host kernel [Dike01]. As shown by Ensim [Ensi03], user Mode Linux [Dike01], [Buch02] and Linux BSD's Jail [Kamp00], along with most other virtual OS technology, adapt virtualization at the system-call level [Bavi04]. Virtualization at the system-call level can be achieved by modification of a kernel code which interacts with hardware [Buch02]. According to Dike [Dike01], linux virtual OS, running on a linux host, can be implemented by a special tracing process using *ptrace* linux system-call; processes in user mode will have their system-calls intercepted and virtualized, but in kernel mode processes should be released from the tracing mechanism and directly run into the host kernel. For a linux virtual OS, hardware interactive assembler instructions in virtual OS kernel (such as interrupt, exception handling, and access functions) can be replaced with signal system-calls, which allows multiple user mode kernels on each virtual device to be hosted on a host kernel [Buch02].

2.4.3 Language-level VMs

Language-level VMs are categorized as ABI VMs, manipulating and supporting both user ISA and "system calls" [Smit01]. The main purposes of language-level VMs are to provide portability, platform independence, and security [Lind97]. These VMs are mainly deployed for use with application programs rather than operating systems; Java virtual machine [Lind97] and Mite virtual machine [Thom99] are examples of language-level VMs. These VMs provide non-native instruction sets, exposing only high-level interfaces to the resources of underlying hardware, while allowing separate and independent designing [Harr01] from the application software.

2.4.4 Queue-based Virtual Machines

The purpose of queue-based VMs is to separate encoding of the implementation from abstract design and to allow for modeling of arbitrary computing systems [Pren00]. Queue-based VM consists of a broker and an arbitrary computable entity; the broker is represented by the enqueueing portion of the incoming queue and the dequeuing portion of the outgoing queue of the computing system, whereby the queues here are generic and any type of queues can be adapted (i.e. FIFO queue, priority queue, etc) [Pren99]. This virtual machine can be used to create systems representing all of Flynn's architectural models (i.e., SISD, SIMD, MISD, MIMD) [Flynn66], allowing set of virtual machines to be connected and composed together for a larger virtual computer [Pren99]. Queue-based VMs are fully generic by decoupling the arbitrary computer system from broker and computable entity, allowing us to enforce various policies on these virtual machines in an efficient way (i.e. security policy, fault tolerant handling, etc) [Pren00].

2.5 Process / VM Migration

Process migration is defined by Milojevic [Milo00] as an "act of transferring a process between two machines" whereby dynamic load distribution, fault resilience, eased system administration, and data access locality are enabled. Process migration at user-level is deployed in many systems, including Condor [Litz92] and MPVM [Casa95], to support cluster computing. Some system-level facilities provided by operating systems (such as inter-process communication) inherently cannot be supported by user-level process migration, whereas object-based process migrations such as those found in Globus [Fost96] and Legion [Grim97] require programming controls on middleware environments without supporting legacy applications for process migration [Osma02]. OS-level VM based process migrations have been suggested [Kozu02], [Sapu02], [Osma02] to support legacy applications with a VM by *capsules* [Sapu02] that can be dynamically instantiated. The advantage of process migration with OS-level VMs is that the latter can encapsulate all volatile execution states of processes, permitting mobile users to suspend their work in one computer and seamlessly resume their work at another computer [Kozu02]. Migrating all states of a running computing environment (including

disks, memory, CPU registers, and I/O devices) across low bandwidth networks has been discussed, with “optimized capsules migration” (copy-on-write disks, which trace only the updates to capsule disks, “ballooning zeros” for any unused memory, hashing, etc) between different VMMs [Sapu02]. The mechanism of a “ballooning technique”, first introduced by [Wald02], requests from the operating system a large number of unused memory pages, which it then “zeros” to help memory states to be easily compressed. Hashing is used to speed up capsule transfer by checking local storage and examines caches; only the data blocks with different hash values would be transferred to reduce the amount data inside *capsule* [Sapu02].

“Remote computational service” architecture, as discussed by [Schm02], can also be used to help “capsule” migration. “Remote computational service”, based on stateless display consoles and cacheable computing sessions, can be connected to session servers via display networks, allowing both the possibilities for active sessions to migrate between session servers, and access to high performance back-end servers which might support clustering and load balancing; users can then access remotely by simply applying low-level, stateless appliance-like consoles while keeping persistent computing sessions [Schm02].

2.6 Resource Partitioning

By using resource-scheduling algorithms [Rajk98], resource partitioning for virtual machines can be achieved by resource reservation, allocation, and scheduling; memory allocation limits can be reserved before a virtual machine boots, with other resources allocation such as CPU and network being specified for each virtual machine (Sugarman, 2001). VMs are not allowed to exceed their share of resources; for example, when a process inside a VM needs to perform computational jobs requiring a lot of memory, this VM alone normally would have to swap and experience a low performance, while other VMs remained unaffected [Dike01]. According to [Wald02], the goals of performance isolation and efficient memory utilization often conflict; a possible solution first introduced by [Wald02] to cope with this is to use an ‘idle memory tax’ (defined as reclaiming more idle pages from inactive VMs), which can specify the maximum fraction

of idle pages requested from a VM, thereby enabling each VM to use a larger portion of its memory without exceeding full share, while efficiently maintaining memory partition.

2.7 Hardware Virtualization

2.7.1 CPU Virtualization

CPU virtualization for a virtual machine can be accomplished by timesharing whereby each guest OS, in turn, gains access to a CPU for a certain period of time allowing the schedule policy of VMMs to control context switching amongst OS-level VMs [Crea81]. According to Barham [Barh03], the x86-based architecture for a traditional CPU at privileged levels can be described as a series of concentric rings, with OS code executing in ring 0 (most privileged), ring 3 (least privileged) is used for application purposes, and rings 1 and 2 are seldom used; for CPU virtualization with VMs in x86-based architecture, VMM should execute in ring 0 while guest OSes (OS-level VMs) should execute in ring 1, thereby preventing guest OSes from directly executing privileged instructions in ring 0, while guest OSes are isolated from running applications. One method of avoiding CPU-virtualization overhead discussed by Whitaker [Whit02], is to have a guest OS issue a virtual instruction (such as an idle-with-timeout) allowing a guest OS to be removed from scheduler considerations until its timer fires or until a signal arrives, thereby helping a guest OS to avoid wasting its slice of physical CPU by executing OS idle loops

2.7.2 Memory and I/O Virtualization

According to Bugnion [Bugn97], “a machine address refers to actual hardware memory, while a physical address is a software abstraction used to provide the illusion of hardware memory to a virtual machine”. Robin [Robi00] demonstrated that an extra level of address translation can be used to both virtualize physical memory and control VM physical-to-machine address mappings; physical addresses can be mapped to machine addresses using the TLB (Translation Lookaside Buffer) of the processor while the VMM can protect and manage the page table for each guest OS. A VMM can use the data structure for each VM to control the mapping of physical page numbers to machine page numbers, as whenever a guest OS issues an instruction to access the TLB or its own page

table, a VMM can intercept this instruction, preventing the VM from updating actual MMU states; Sugerman [Suge01] first introduced ‘shadow page tables’ which can be maintained by a VMM for a processor’s TLB to perform machine-to-physical address mapping, avoiding additional overhead during virtual-to-machine address mapping.

For the purposes of I/O virtualization, Robin [Robi00] demonstrated that a VMM should intercept each VMs access to I/O devices and forward them to physical I/O devices in order to virtualize the latter; during this process, one special device driver for each type of device can be used (rather than the real device driver in every I/O device) by first introducing a “monitor call” which directs all command arguments to the VMM into a single trap for simplicity and efficiency.

2.7.3 Full Virtualization

According to Creasy [Crea81], traditional virtual-machine monitors (VMMs) provide each guest OS (VM) with a full hardware virtualization; virtual hardware exposed to each VM should be functionally the same as the underlying machine so that a virtual-machine monitor can host unmodified multiple operating systems while giving guest OSes (VMs) the illusion they are running directly on physical hardware. For a full hardware virtualization, all hardware-specific instructions by VMs should be intercepted by a VMM; whenever guest OSes (VMs) or applications execute privileged instructions (including hardware instructions) by traps, VMM should intercept these traps prior to a VM interaction with the hardware, and whenever VMs are required to execute non-privileged instructions (such as simple arithmetic operations), those non-privileged instructions are allowed to directly execute on the CPU without VMM intervention [Robi00]. Whitaker [Whit02] pointed out that traditional mainframe hardware, especially the processor, was designed to be virtualizable, but the Intel IA-32-based processor architecture is not completely virtualizable; some x86 sensitive instructions, which might affect the states of a VMM or other VMs, were not trapped in user-mode, for example, some x86 instructions (ex. `pushl`, `popl`) access the interrupt-enabled flag in this mode without being trapped. Therefore, full virtualization is not possible in the Intel IA-32-based processor architecture; for full hardware virtualization on Intel-based architecture,

inserting manual traps by binary rewriting (thereby emulating full virtualization) was one solution suggested by [Suge01].

2.7.4 Paravirtualization

According to Whitaker [Whit02], traditional VMMs with full hardware virtualization demonstrated performance drawbacks because large amounts of memory are consumed by each VM in order for the latter to access its own copy of resources and devices. Xen [Barh03] and Denali [Whit02] systems apply ‘paravirtualization’, which is the virtualization of a subset of the processor’s instruction set with specialized virtual devices to enhance performance. A ‘paravirtualization’ system replaces hardware interrupts with its own event system to provide control transfer between VMM and VMs; for example, Xen systems allow VMs read access to page tables, but a VMM intercepts the write access for updates from VMs by Xen’s trapping mechanism [Barh03] to enhance its performance. An ‘isolation kernel’, similar to a VMM, can also be used as a ‘paravirtualization’ system; the Denali isolation kernel (which provides a simplified interface of an underlying architecture) removes deprecated and rarely used machine instructions and modified some instructions (such as nonvirtualizable instructions in the x86 architecture), then adds particular virtual instructions (thereby enabling guest OSes to be directly executed onto the physical processor in some cases) for the isolation kernel’s instruction set [Whit02].

Another example of ‘paravirtualization’ in Denali systems is shown by its replacing of complex BIOS bootstrap functionality of guest OSes with the simple procedure of a VMM loading a VM image into memory [Whit02]. For the purpose of minimizing I/O virtualization overhead for each VM, a Denali system drastically reduces the number of I/O devices (supported by guest OSes), keeping only those found in a typical system, such as a network interface card, serial device, keyboard, timer, and a console; thousands of the modified guest OSes (VMs) can be hosted on an ‘isolation kernel’ by this ‘paravirtualization’ approach [Whit02]. Using these methods of ‘paravirtualization’, the isolation kernel should be resident in physical memory, while VMs should be paged on demand; whenever page fault is taken by the VM, the isolation kernel verifies the virtual

address, allocates new page tables, and initiates a read action from the VM's swap region [Whit02]. For improved performance, an 'isolation kernel' should mandate each VM's access to a subset of virtualized address spaces, with the kernel itself being mapped into those address spaces inaccessible to VMs, thereby avoiding excessive TLB (Translation Look-aside Buffer) flushing onto VM/VMM crossings while providing the means for the sharing of memory between VMs [Pete02a]. An isolation kernel can also choose a select number of active VMs to be in memory; as the remaining VMs are swapped to second storage, this process periodically redistributes physical memory from inactive to active VMs [Whit02]. However, if existing native operating systems are to be used in either Xen or Denali systems, drastic porting efforts might be required [Bavi04].

2.8 Network Virtualization

2.8.1 Programmable Network

The goal of a programmable network is to simplify the network services for their deployment [Camp99a], [Camp99c]. As stated in [Camp99a], a programmable network decouples control software from communication hardware to virtualize network infrastructures. According to Campbell [Camp99a], several prototypes of programmable networks have been suggested by a number of research groups. One class of programmable network suggested by Campbell [Camp99c] is a *spawning network* (whereby a child network operates on a subset of its parents' network resources, independently performing despite the limitations in their parents' resource and partitioning models), allowing the creation, deployment, and management of new network topologies, through the virtual network operating system's "life cycle" (composed of profiling, spawning, and management) process. A virtual network should be defined as a "profiling process" (the selection of topology from the parent link and nodes and the specifying of resource requirements for virtual links, including bandwidth and capacity) before spawning [Camp99c]. The main procedure in the "spawning process" is the dynamic instantiation of profiling scripts (such as setting up topology), the allocation of resources, the creation of virtual network components, and the bootstrapping of network services [Camp99c]. The "management process" in the virtual network

operating system's life cycle depends upon the "per-virtual-network" policy, which allows the management, control, refinement, and monitoring of virtual network resources [Camp99c]. Campbell [Camp99c] demonstrated that programmable network can be used to architect, compose and deploy virtual networks by his example of spawning networks.

2.8.2 Network Components Virtualization

Physical network devices can be abstracted as distributed computing objects such as virtual switches [Merw97b], virtual routers [Camp99b] and virtual ATM [Merw97a], [Merw97b] in a virtual network [Camp99a]. According to Jiang [Jian03c], virtual network interfaces can be dynamically created, configured or deleted even when the virtual machine is active; when a new request for adding/deleting a virtual network interface arrives, a virtual machine accommodates this new request by renewing VM kernel data structure after proper authentication. A virtual switch is also created for each virtual LAN, with packet forwarding then performed by the former at data link (layer 2) level; the Unix/Linux *poll* system-call can be used to emulate a physical switch, whereby a UDP Daemon polls the arrival of data and manipulates forwarding or dropping incoming requests [Jian03b]. When the proper VM connect request for virtual LAN arrives, a new port can be allocated for the virtual machine by a virtual switch, enabling a physical connection to be established between the virtual machine host and the virtual switch, as shown in the figure below:

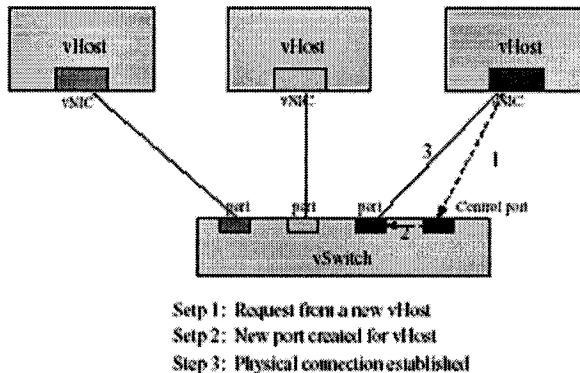


Figure 2 Virtual switch and steps of port creation [Jian03b, page : 5]

Basic differences between a physical switch and a virtual switch include the lack of hardware constraints to the number of ports possible for the latter, as well as the capability of virtual switches to use various packet queuing/forwarding policies which can be dynamically adapted for loss rate, bandwidth, congestion, and delay [Jian03c]. As a result, network component virtualization can avoid the need for a restart when it is used in a dynamic, adaptive VM overlay network [Jian03b].

2.9 Model Driven Software Engineering

2.9.1 Model and Metamodel

In the context of model driven software engineering, a model of a system is defined by Kleppe [Klep03] as “a description or specification of (part of) a system and its environment for certain purposes, and it should be written in a well-defined language; a well-defined language was defined as a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer”. According to Nytu [Nytu02], the metamodel is by itself another model to define other models; the metamodel identifies possible structures and the meaning to elements in a model. The Object Management Group [OMG01] proposed four modeling layers, called M0, M1, M2, M3 (see table 1). Each layer provides a service to its upper layer and serves as a client to its lower layer [Estr01].

Table 1 : Four Modeling layers of the OMG [OMG01]

Layer	Description
M3:meta-metamodel	The infrastructure for a metamodeling architecture. Defines the language for specifying metamodels.
M2: metamodel	An instance of a meta-metamodel. Defines the language for specifying a model.
M1: model	An instance of a metamodel. Defines a language to describe an information domain.

M0: user objects (user data)	An instance of a model. Defines a specific information domain.
---------------------------------	--

As denoted in Table 1, the meta-metamodel layer (M3) defines the language for specifying metamodels, and the metamodel layer (M2) defines the language for specifying models. The model layer (M1) defines the language for specifying information domains, and the user objects layer (M0) contains user objects and user data respectively [OMG01]. According to Kleppe [Klep03], any number of levels could potentially be used. However, instead of defining an M4 layer, the OMG mandates that all elements of layer M3 be instances of the M3 layer itself. The OMG [OMG01] proposed Meta-Object Facility (MOF) as a standard M3 language, allowing modeling languages (e.g. UML, CWM) to be instances of the MOF.

2.9.2 Model Driven Architecture

Model driven architecture (MDA) is a framework for software development made by Object Management Group (OMG) and has three primary goals: they are portability, interoperability, and reusability [OMG01]. The MDA process defines three steps (see Figure 3).

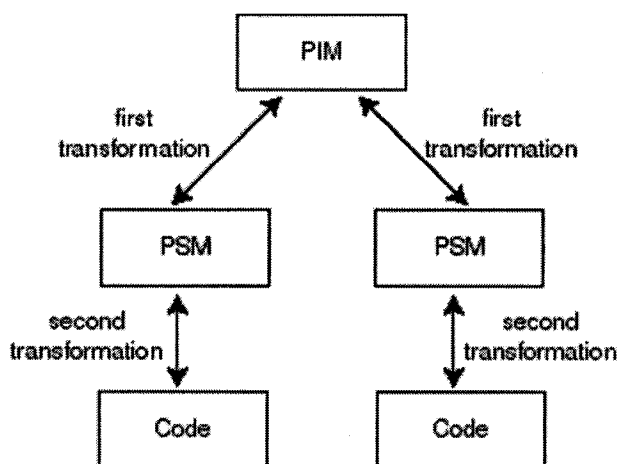


Figure 3 MDA Process [Klas04]

The first step is to build a model with a high level of abstraction that is independent of implementation methodologies. The model with a high level of abstraction is called a Platform Independent Model (PIM). The second step, the PIM is transformed into one or more Platform Specific Models (PSMs). A PSM specifies the model in terms of the specific implementation technology (e.g. EJB model). The final step is to transform a PSM to code (e.g. Java, SQL). The MDA allows software development procedure to be mainly focused on producing a high and abstract level of the system. Thus high level models should be written in a standard, well-defined language (e.g. Unified Modeling Language (UML) in combination with the Object Constraint Language (OCL)), in a manner that is consistent, precise, and contain enough information on the system [Klas04].

2.9.3 Software Component Modeling

According to [Wiki], a software component is the “software technology for encapsulating software functionality, often in the form of objects (from Object Oriented Programming), in some binary or textual form, adhering some IDL (interface description language) so that the component may exist autonomously from other components”. The component models can be specified as the four modeling viewpoints. They are interface models, static behaviour models, dynamic behaviour models, interaction protocol models [Rosh03]:

1. Interface models specify the access points that allow a component to interact with other components in a system.
2. Static behaviour models shows the functionality of a component in a discrete manner, i.e., at a particular instance during the system’s operation.
3. Dynamic behaviour models provide a continuous view of a component, thereby describing different states in its execution.
4. Interaction protocol models provide an external view of a component, thereby describing its interactions with other components.

Bilke [Bilk02] pointed out that components do not exist independently of component platforms where runtime environments are provided. In this sense, to develop components is subject to a dedicated component platform. Ziadi [Ziad02] proposed a “platform independent component model”, which can be considered as the PIM (Platform Independent Model) of the MDA, allowing the modeling of software components to be independent of any platform environment; OCL meta-level constraints and mapping rules (i.e. generating source class skeletons and IDL files) can be useful to map platform independent component models to platform specific models.

2.10 Related works

2.10.1 Grid Computing on OS-level Virtual Machines

By using a ‘VM life cycle’, the mechanism of grid computing on OS-level virtual machines have been illustrated. Grid computing on an OS-level virtual machine involves a physical, virtual machine O/S image, application image, and user data server [Figu03]. The steps of a ‘VM life cycle’, in grid environments, are as follows:

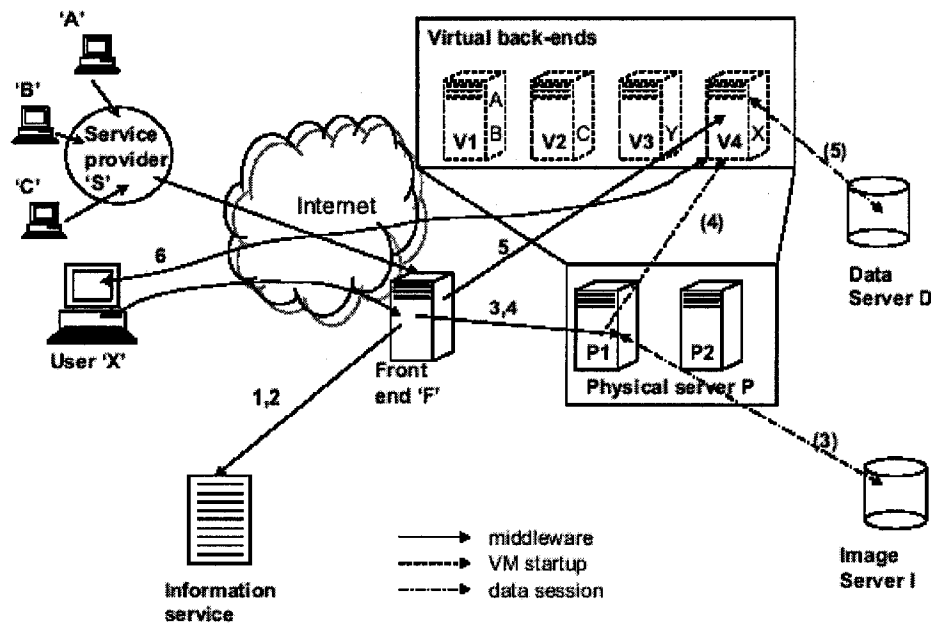


Figure 4 Architecture for a VM-based Grid Service [Figu03, page: 12]

According to Figure 4, user X (or Service Provider S on behalf of users A, B, and C) queries the availability of resources for use by information services such as Globus MDS, [Fost96] or GIS [Alle01], using middleware Front end 'F'. Then, grid middleware establishes a data session between physical server P and image server I, either by GridFTP [Fost02] or on-demand transfers [Figu03]. When the data session is established between physical and image server, the former downloads images from image server and is then able to reduce transfer delay later by caching the VM state; once the download is complete, a physical machine allocates a slice of its resources for a VM image [Figu03], then instantiates a VM while providing it with an IP or virtual Ethernet address [Sund03]. Virtual back-ends are groups of VMs which are mapped slices of physical machines; data sessions for application downloads are then established between operating systems inside VMs and the application server [Figu03]. These transfers can also be achieved by on-demand transfers. Users can then execute applications with SSH or Globus GRAM [Fost02] either by interactively using remote display protocols such as VNC [Rich98], or batch modes [Figu03].

2.10.2 Virtual OS based Distributed Systems

The hosting of application services by virtual OSes as a distributed-system utility has been suggested [Jian03b]; specifically, Service-On-Demand Architecture (SODA) hosted upon service Hosting Utility Platforms (HUP), providing on-demand creation of application services in distributed systems' environments. These application services, including guest operating systems (virtual OSes), are dynamically created and automatically bootstrapped as a group of virtual service nodes, and each virtual service node is represented as a virtual machine, providing administration isolation in addition to fault and attack isolation [Jian03b]. The components of Service-On-Demand Architecture (SODA) are middleware entities SODA Agent, SODA Master, SODA Daemon, and Service Switch; initially, the Application Service Provider (ASP) requests service creation to the SODA Agent with a resource requirement, after which the SODA Master checks whether the resource requirement of ASP can be satisfied with the HUP resource availability [Jian03b]. If the resource requirement can be acceptable on HUP, SODA Master consults the SODA Daemon, with the latter downloading application service

images and bootstrapping virtual service nodes; after the service bootstraps, Service Switch will accept client requests and redirect to the appropriate virtual service node [Jian03b]. A similar approach is that available with the Denali [Whit02] and Xenoserver [Hand03] projects, both providing isolation between internet services on shared hardware in distributed/grid environments. These application services can complement web service based grid platform [Fost02] due to its resemblance to service-oriented architecture [Jian03b]

2.10.3 VM TestBed

According to Jiang [Jian03b], VM overlay networks can be deployed to test and monitor underlying physical networks and applications running the VMs. Virtual-machine monitor-based overlay supports distributed virtualization (with each node able to provide simultaneously-running multiple services in a multiplex manner) allowing each application to be run as a part of the overlay, and not globally scheduled to run [Pete02b]. In Virtual Internetworking on Overlay Infrastructure (VIOLIN) suggested by [Jian03b], network components such as routers, switches, and end-hosts can be virtualized on top of overlay infrastructure to be user-configurable on demand, easily arranged for different testbeds associated with VM based distributed systems. The components of this architecture consist of virtual end-hosts (i.e. virtual machines in physical hosts) and virtual routers (i.e. virtual machines with multiple virtual interfaces, having the capability of forwarding between each virtual LAN) [Jian03b]. Virtual LANs can be organized by individual virtual switches which connect multiple virtual end-hosts, and are responsible for packet forwarding (at the data link layer); this architecture creates a VM network for the various services of distributed systems with no modifications of the real internet infrastructure, making the testing of a VM network for different services of distributed systems easier [Jian03b]. The figure 5 illustrates relations between overlay infrastructure of virtual components, and underlying internet. Sundaraj [Sund03] first suggested Physical, VMD (Virtual Machine Daemon), and VM layers for the monitoring and testing of VM networks; the first is an underlying IP network, while VMD layers are an overlay in this architecture and able to manage VMs, monitoring both the resources provided by underlying physical networks and the resources requested by VMs in VM layers.

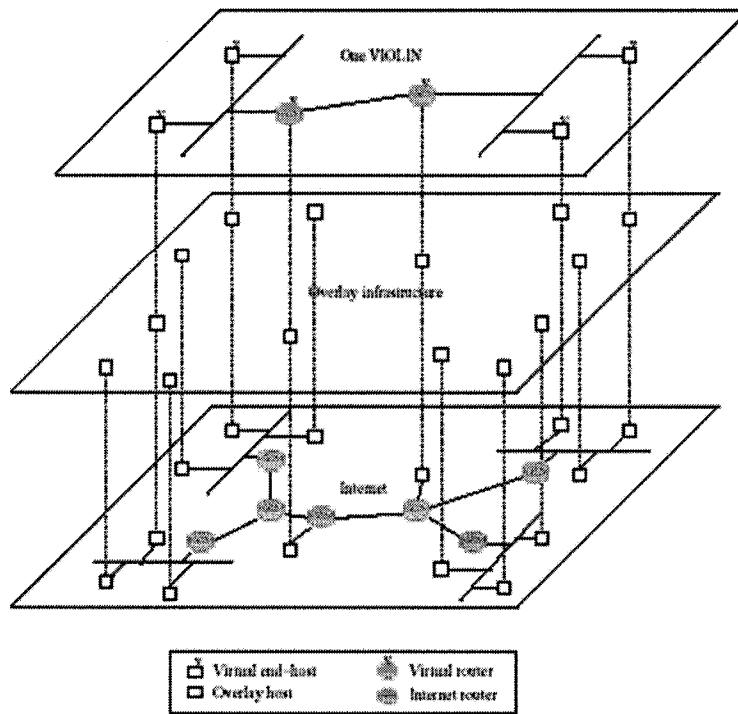


Figure 5 Virtual Internetworking on Overlay Infrastructure [Jian03b, page: 2]

According to Jiang [Jian03c], the major steps for testing VM networks are as follows: first, specified VM and VM networks are required by using a well-defined script language; second, the logical entities in the testbed should be mapped onto virtual machines in VM networks; third, VM networks should perform virtual-node and virtual-topology creation; finally, the test of distributed systems or grid services (either batch-oriented or interactive) can be conducted with experimenters monitoring and managing VM networks at run time. By monitoring the VMD layer, Sundaraj [Sund03] demonstrated that it is possible to adapt communication and computation behaviour of VMs in the VM layer, allowing VMD routing rules and topology to be changed for the purpose of efficiently migrating VMs and/or deploying various services in distributed systems or grid.

3 A PROPOSED VIRTUAL MACHINE (MVM)

As the first step towards developing a logical framework that matches the vision of our design goals, we have devised a Multi-layer Virtual Machine (MVM) (see Figure 6). We propose to build a large, logical, high performance virtual computer on existing systems. Therefore, our strategy is to keep underlying systems intact, without any dramatic changes to the host OSs. Studies on hardware virtual machines to enable network reconfiguration are currently being conducted by FPGA (Field Programmable Gate Array) research groups [Scha04]. However, this model does not fit our scalable virtual machine design goal, as it is mainly used for homogeneous platforms with homogeneous computing techniques. As stated in Chapter 1, the MVM is designed to overcome existing grid middleware difficulties. The proposed MVM middleware is intended to provide an alternative method to traditional grid middleware methodologies or complement them. The MVM¹ consists of three layers: *OS-level VM*, *Queue VMs*, and *Components VMs*. The *OS-level VM* is a virtualized operating system that can be migrated within the network. The *Queue VMs* consist of broker and processing entity. The *Components VMs* allow us to reconfigure VM interface logics at run time to control VM itself, for a wide variety of job, resource, and connectivity models. Detailed descriptions of each different layer will be presented in Section 3.2-3.4.

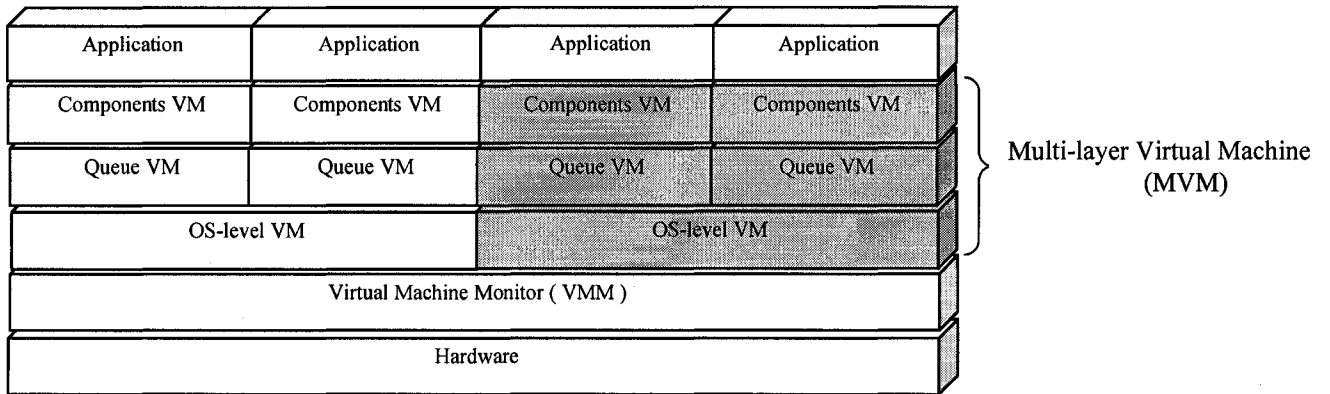


Figure 6 Multi-layer Virtual Machine Architecture (type (a))

¹ type (b) and type(c) MVM is discussed in Section 3.5

3.1 Building a virtual computer using MVMs

This paper presents a design for a large distributed and parallel reconfigurable virtual computer for a virtual laboratory, on heterogeneous platforms. The schematic diagram for the virtual computer using MVMs is shown in Figure 7. One of our design goals stated in Chapter 1 is for decentralized maintenance. We take advantage of existing super peer based P2P architecture. The experimenter joins a virtual community and then retrieves nodes information. Then one or more strong peers with a high bandwidth can be selected as super peers, either statically or dynamically. The processing element of the super peer will act as a control processing element, while the processing elements of the remaining nodes will act as general processing elements. The role of the control processing element is to control the general processing elements. According to our interface logics, the control processing element will schedule general processing elements, sequence jobs, and manage the connectivity for the general processing elements. Actual computations for the experiment will be conducted by general processing elements. Interface logics consist of logical, reconfigurable components. Our logical components, such as connectivity and sharing rules, will be further mapped to physical connectivity and sharing rules. In many existing systems, the configurations are too coupled with underlying architecture, so we had to rewrite every different configuration for every job and resource scenario. However, virtual laboratory users might not need to be aware of such details of underlying architecture. Our logical components in the *Components VM* will provide a logical layer and raise the level of abstraction, providing a simplified view of the system.

The VM interface controller in Figure 7 will buffer incoming data according to different queue types (i.e. FIFO queue, priority queue, etc) and set policies according to the interface logics. The VM decoder allows us to obtain actual data in different architectural types by referencing interface logics. In contrast, the VM encoder allows us to convert data, instruction, and protocol for a specific use. In our MVM design, there are two kinds of memory types: VM external memory and VM internal memory. The VM external memory is the super peer's shared disk space or backend data storage, while the VM internal memory is the disk space for the general processing element.

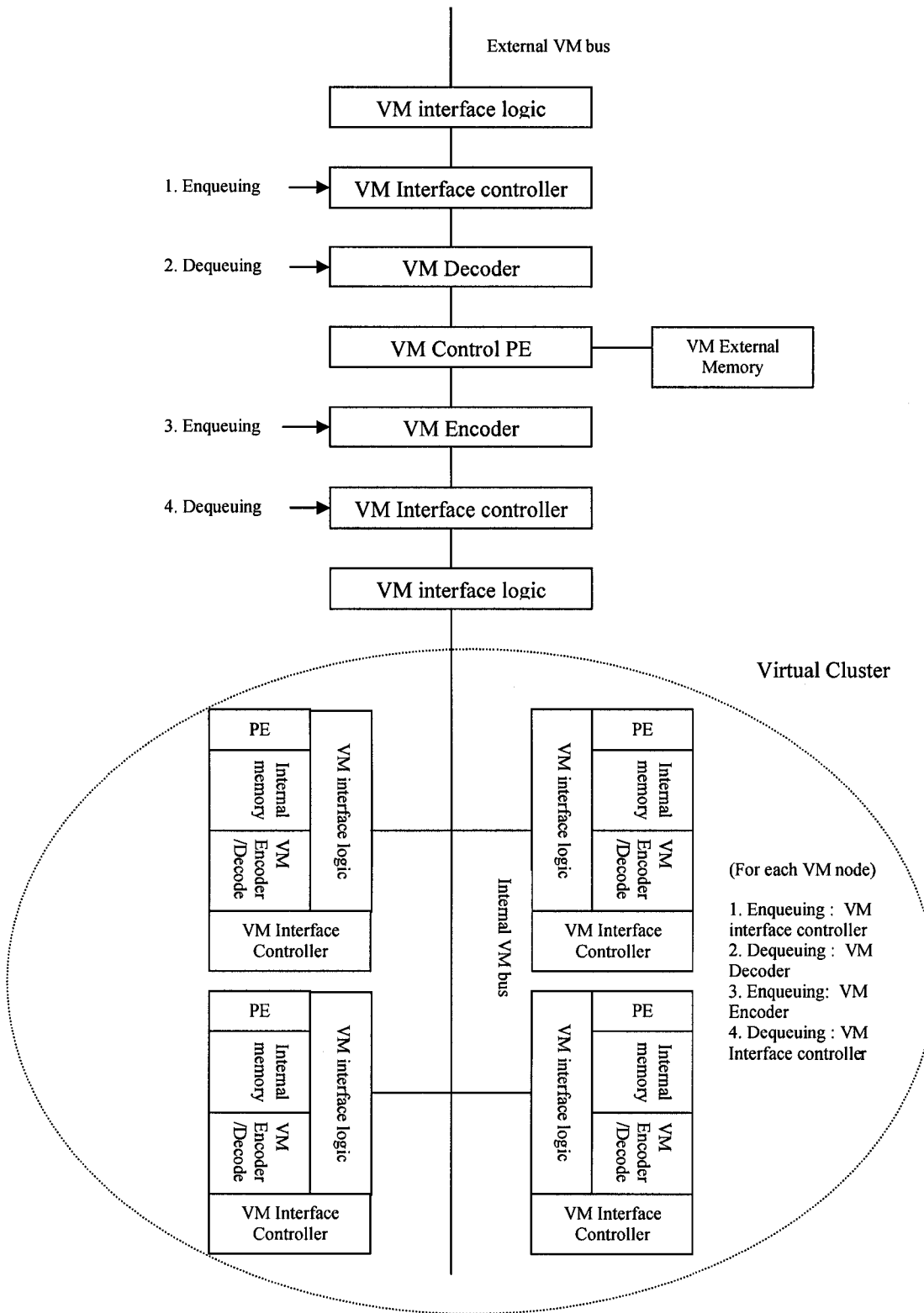


Figure 7 Virtual Parallel Computer using MVMs

As the *data staging* technique [Flin02] reduces latency by moving data to a nearby surrogate, we prefetch often used files, in the super peer's external memory, in order to improve the performance of the virtual cluster. We refer to a group of MVMs as the virtual cluster in that it is dynamically organized, for the specific experiment by the research communities in the virtual laboratory, and it is not confined by physical locations. Once the community is organized, it can be a Virtual Organization (VO) in grid jargon, as it has the same sharing rules across the nodes in the community. In Figure 7, we emulate real data communication paths for the MVMs with our external VM bus and internal VM buses. However, we can replace the internal VM buses with other network communication models, such as the ring, and the star, without loss of generality.

3.2 OS-level VM layer

We take advantage of existing OS-level VM technology to satisfy parts of our design goals, especially fault/attack isolation. What if one experiment generates a fault and freezes host OS? What if the host OS is compromised by an attack during the experiment? In our MVM design, distinct OS-level VMs can be multiplexed on the Virtual Machine Monitor (VMM). In addition to the isolation feature of the OS-level VM technology, the OS-level VM can be used to set up a virtual network, which allows the setting up and testing of experimental services. Let us consider the following scenario.

1. Alice participates in two research projects, for example, neural network project “a” and biochemistry project “b”.
2. Alice joins community “a” and community “b” for each project, in the virtual laboratory.
3. Alice wants to use her own real machine, as if two independent networkable machines are running on her machine, with its own processor, sharing and resource management rules.

In this case, Alice can allocate two OS-level VMs on VMM for each community. Alice is then able to customize each OS-level VM for each community with its own specific sharing, connectivity, jobs, and resource management rules as if two autonomous

dedicated host machines are running for each community. We may consider downloading and installing the OS-level VM when we join a community, and simply discard it when we leave the community. Current ongoing projects, such as VMPlant [Krsu04] and SODA [Jian03a], shows that the on-demand, dynamic instantiation of VMs can be incorporated into the grid resource and grid data management framework. In a similar way, we can store a wide variety of OS-level VMs in the virtual backend, and then retrieve and assemble them with the Queue VMs by utilizing existing grid middleware methods.

The OS-level VM in the MVM can also facilitate the underlying system to be maintained in a partitioned way. All manipulations inside the OS-level VMs do not affect the configuration values in another OS-level VM on the underlying system. We often find that multi-user systems restrict the privileges of users for system protection. They also restrict some system facilities, such as the logging and monitoring the programs at the system level. In certain circumstances, a user or a group of users need some system facilities to run and monitor their experimental programs. Thus, in this situation, the OS-level VM technology is a viable choice for them to have their own OS on the underlying system, while keeping the OS-level VM from damaging the underlying system. Another advantage of adapting the OS-level VM as a building block of the MVM is that we can migrate the processes in the OS-level VM including the OS-level VM itself. As we stated in Section 2.5, when a virtual machine suffers from a fault or failure, we can use VM-based checkpoint technology [Schm02]. In this case, we have two choices: the first choice is that we migrate a group of *Queue VMs* in MVM to another node, without migrating the OS-level VMs; the second choice is that we migrate the whole MVM, including the *Queue VMs*, to another idle node. The “check-pointing jobs” [Buyy99] are types of jobs that periodically save their status to the files system, and thus can be aborted and resumed anytime. When they are restarted, they resume execution from the last checkpoint. The “checkpoint-jobs” can facilitate the migration of jobs between the MVMs. However, additional efforts are required to modify usual jobs to the checkpoint jobs.

3.3 Queue VM Layer

Queue VM consists of a broker and an arbitrary computable entity. The arbitrary computable entity is categorized by the control processing elements and general processing elements in MVM. The key part of the *Queue VM* is the broker that provides all of the mapping, between a computing system's external input and output interfaces, and their corresponding internal computable entities [Pren99]. We further divide the broker in the *Queue VM* into our VM interface logic, VM interface controller, and VM encoder/decoder. Our design of the *Queue VM* in the MVM is fully generic and flexible, so that we can (re)configure and control how to interface an arbitrary computer system. We advocate that "enqueueing" and "dequeueing" are the most common characteristics of any computer systems available. Any computer system should enqueue and dequeue its data including instructions. The actual computation is dependent how to take items in the queue, and how to decode these items. Once we can configure how to do this process at run time, it will give us a chance to build a flexible and scalable system, on the arbitrary computing systems. Furthermore, it will give us strong and versatile features of system design, which allows us to accommodate a large number of different architectures. Suppose that computational jobs are fed into the group of *Queue VMs*, consisting of arbitrary architectures. As shown in Figure 7, the "enqueueing" and "dequeueing" portions of the VM can extend to a group of computer systems. We do not know how the processing elements are alike. However, we are able to look at the queue, pick up items in the queue by using the interface logics, and then apply various rules to these items. These rules contain the information on how to decode the items, and how to run the items in a group of nodes. When a group of *Queue VMs* are organized under the certain sharing rules for the specific experiment, it provides a Single System Image (SSI), as if one broker and one processing element are running to the external user. In this case, the broker can be regarded as the super peer VM node, and the virtual cluster in Figure 7 can be regarded as the processing element. When we look at the inside of the SSI in Figure 7, we find that it follows the Flynn's MIMD (Multiple Instruction and Multiple Data Stream) architecture [Flynn66], in that each connected VM can provide either instruction stream, or data stream, for the parallel computation. We might also have several queues for the

enqueueing process in the *Queue VM*, so that we can assign different queues to different jobs, allowing each job to be run concurrently in each queue in SSI. Thus, we can simulate the parallel computer architecture by organizing a group of *Queue VMs* in the MVMs.

3.4 Components VM Layer

Although our VM middleware shares the same key problems (resource discovery, allocation, management, authorization/authentication, policy enforcement, etc) with the existing grid middleware, our approach to cope with the problems is to “configure and customize”, by using lightweight component-based technology. Another advantage of the component-based technology is the reusability. Once we register the connectivity, resource and job profiles for a certain experiment, we might use the same profiles with different parameters for later use. The *Components VM* layer, in our MVM, allows us to reconfigure the virtual machine itself. When we organize a virtual cluster for the experiment, we might apply different jobs and resource usage scenarios, for later experiment. When an experimenter sends a job and its header to the broker node, the broker node (super peer) first sees the job header. The job header includes the component names of the job, resource and connectivity profiles. The broker node then retrieves each component, and organizes the virtual cluster by the “connectivity profile” component. The “connectivity profile” component specifies the logical topology, along with the stage information for the pipeline applications. The “resource profile” component includes resource scheduling policy, and sharing policy. The resource scheduling policies includes First Come First Served, Select-Least-Loaded, Select-Fixed-Sequence, etc [Buyy99]. The resource sharing policy describes logical sharing rules across the virtual cluster. The “job profile” component includes the resource requirement and the resource limitation for the job. The above profile also includes timeout value, queue types (e.g. FCFS, SJFS, priority), number of queues, queue sizes, job sizes, byte orders (little endian, big endian), etc. The resource requirement module describes the required resources, such as maximum and minimum number of nodes, memory space, bandwidth, etc. The resource limitation module describes the limitation of the resources for the job, such as timeout for the resource usage, minimum CPU speed (MIPS), minimum memory, and minimum disk

space. When we first conduct the experiment, we can store our configuration values into a file. These configuration values will be default values, for later use. However, when we conduct our next experiment with a slight change of the configuration, we can specify the changed values into the parameters or update and retrieve the configuration components. We do not have to reboot our MVMs in this case. This runtime reconfiguration configuration capability facilitates the reduced maintenance cost and time. We further set the interface logics by using the *Components VM*, which determine the behaviour of the broker in the MVM. The policy enforcement will be established at the broker as well. The broker is the key module of *Queue VM*, represented by the enqueueing portion of the incoming queue, and the dequeuing portion of the outgoing queue, of the computing system. It provides a uniform interface to access, while hiding the complexity of the underlying system. Our plug-in *Components VM* determines how we manipulate the broker, and how we adapt and translate it, into a particular usage scenario. In that sense, the *Components VM* provides a semantic for the *Queue VM*. The “policy-based reconfigurable components” are described in Section 4.3.

3.5 The type (b) and type (c) MVM

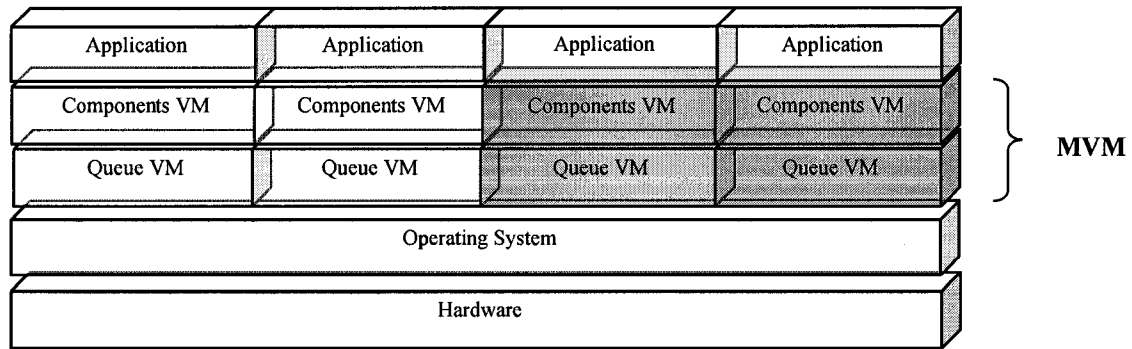


Figure 8 Multi-layer Virtual Machine Architecture (type (b))

Whenever we virtualize something in a computer system, we can often reduce management cost by providing a predefined interface for virtualization. For instance, if we virtualize the OS, the management cost can be lessened. As our virtual laboratory aims to provide a certain degree of self-configuring and self-management capabilities, the virtualized OS is also an essential feature. However, there are trade-offs for OS

virtualization. The trade-offs are we have to modify the underlying system, and the additional performance cost that is required to map virtual operations to system level operations. The advantage of the type (b) MVM is that we do not need to modify the underlying system. Although the OS-level VM layer does not exist in the type (b) MVM (see Figure 8), it provides the basic functionalities of the MVM, such as reconfigurability, and flexibility. However, fine-grained resource customization and partitioning are not supported in the type (b) MVM. In type (b) of MVM, *Queue VMs* provides a uniform interface to a group of heterogeneous systems, thereby allowing the collaboration of a wide variety of distributed, heterogeneous systems. The *Queue VMs* are also utilized as virtualized queues for virtual networks. Once the connectivity information has been given, those virtualized queues are dynamically connected and organized, providing a virtual network for heterogeneous processing elements.

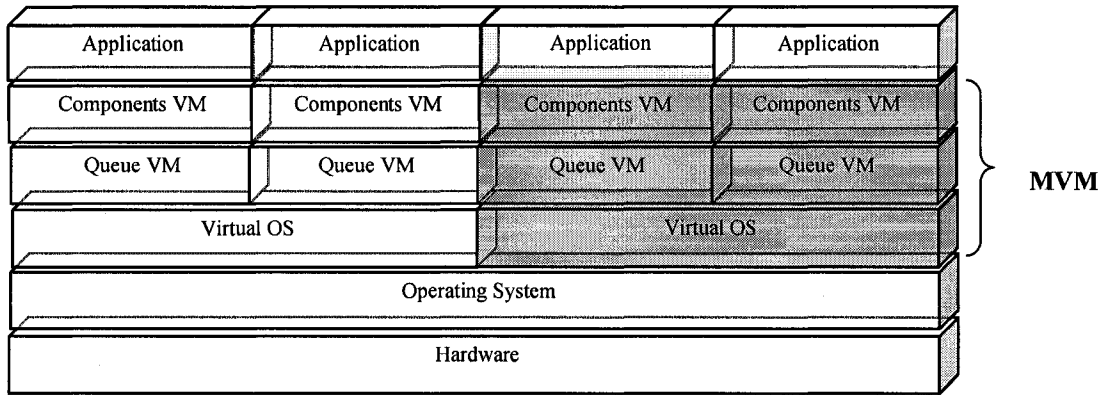


Figure 9 Multi-layer Virtual Machine Architecture (type (c))

The type (c) MVM uses a user-level OS (virtual OS) technology, whereby the virtual OS kernel and its processes run as processes on the host kernel [Dike01]. Thus, the resource partitioning and customization are not fully supported. The type (c) MVM is not required to essentially modify the underlying operating system. It can be dynamically created on-demand, and automatically bootstrapped, while providing administration isolation, in addition to a certain degree of fault and attack isolation. Thus, it can be used for utility computing architectures and service hosting environments [Jian03a], in addition to the virtual laboratory framework.

3.6 Application Model

The application model is not particularly defined in the MVM. Program developers or experimenters should choose the application model required for the specific experiment. Prior to an experimenter sending a job to the broker node (super peer), the experimenter sets basic configurations by using a user interface. The basic configurations include the application profile, job and resource profiles, and connectivity profile. The experimenter can reuse and edit those configurations, by utilizing software component technology, in which the components are stored in the backend server. The user interface displays the information of each component that was previously used. After the experimenter edits properties of the above profiles, the experimenter should update it or store the profiles using different names to the backend server. Once the experimenter finishes setting up the configurations, the user interface program attaches a header to the job. The header consists of the names of each configured component. Figure 10 describes the application model in the MVM, consisting of the job and its header. In Figure 10, the application profile includes the encoding and decoding information of the job, and the management policy. The encoding and decoding information specifies whether the payload is a job or data. It also describes data types and language types, such as Java, C, C++, etc. The management policy is used to encapsulate and decapsulate the job header, additionally it handles the congestion and fault situations. The management protocol should be supported by the *Queue VM* in MVM. The situation whereby major congestion often occurs in the MVMs is where the number of the general processing elements is too high, compared with the capacity of the control processing element. In this case, we need to partition the virtual cluster, into two or more sub clusters. Congestions and fault situations will be discussed in Section 4.2.2 and Section 4.3 respectively.

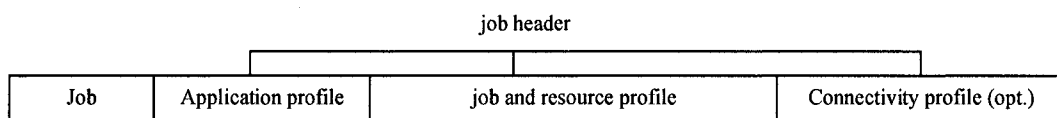


Figure 10 Job and its header for MVM application

4 A PROPOSED FRAMEWORK FOR VIRTUAL LABORATORY

The proposed virtual laboratory has three different layers to provide dynamic and flexible test environments, for a wide range of research communities: they are the virtualized resources, virtualized networks, and policy-based reconfigurable components. For scalability, the virtual front-ends and the virtual clusters in virtualized resources are organized in a decentralized way. However, a pure decentralized system is known to have several drawbacks, such as bandwidth overuse by message flooding, and maintenance difficulties [Dasw02]. We advocate using the “super peer based P2P system” for aggregating virtual resources. It has super peers for the delegation of each peer group. The virtual community has one or several peer groups, whereby each peer group has a super peer. The super peer in each peer group acts as the virtual front-end, in that it provides a group of peers with virtual back-end information. This data includes URI location of the virtual backend, and the ways on how to retrieve information from the virtual back-end. We adopt the *k-redundant* super peers’ system for our virtual laboratory, whereby the *k-redundant* super peers are used for system availability and redundancy [Yang03]. We use *2-redundant* super peers, for the virtual front-end in the virtual laboratory. The redundant super peer node periodically checks the heartbeat of the super peer node, and updates the connection status of the super peer. The “failover” and “failback” are popular methods of the cluster recovery model [Buyy99], and we take advantage of this process between the super peer and the redundant super peer in the virtual front-end. The redundant super peer will also be used to share the load of super peer. A virtual switch will be maintained for each super peer (See Figure 2). When the failure occurs, the redundant super peer node takes over some ports for the virtual switch in the super peer. The super peer then forwards the requests from ports that have been taken over, to the redundant super peer.

The descriptions of virtualized resources are presented in Section 4.1. The descriptions of virtualized networks and policy-based reconfigurable components in the proposed virtual laboratory framework are presented in Section 4.2 and Section 4.3 respectively.

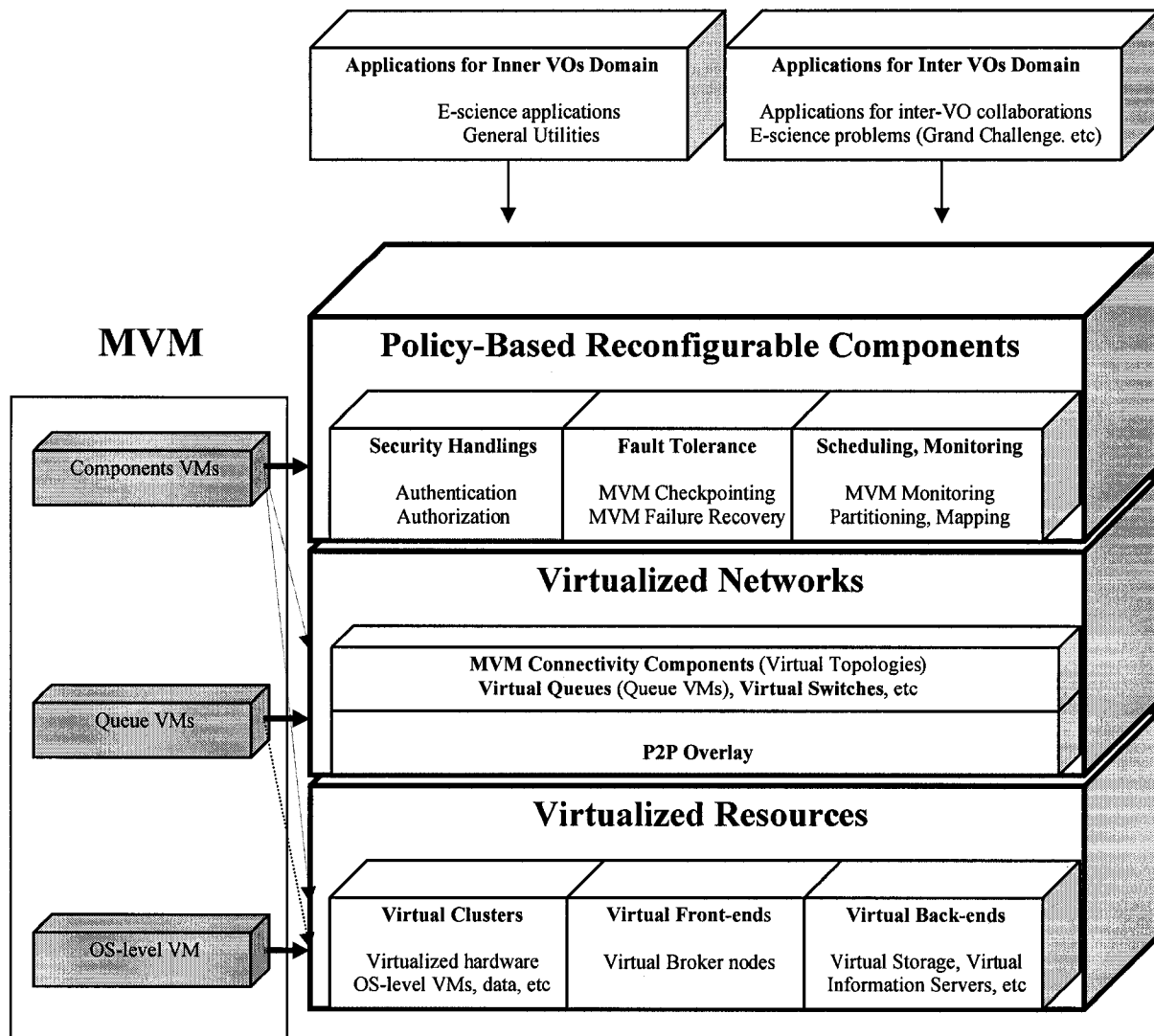


Figure 11 Proposed Virtual Laboratory Framework

4.1 Virtualized Resources

4.1.1 Virtual Back-Ends

The virtual back-ends in our virtual laboratory design consists of several remote servers that provide the virtual front-ends and virtual clusters with our VM image, resource discovery service, bootstrapping service, components storage, etc. There are two choices when using the virtual back-ends for the virtual clusters. One method is to use a virtual front-end (super peer) as a data surrogate for cache, such that a super peer retrieves data

from the virtual back-end in advance, and then disseminate the data to the virtual cluster. The other method is that the virtual front-end only provides the virtual cluster with references for the virtual back-end. In this case, the virtual cluster has to retrieve the actual data from the virtual back-end by using the references of the virtual front-end. The trade-off of performance improvement on the first method is we have to manage cache consistency protocol [Wu04], in addition to policy enforcement, when the maximum capacity of the super peer is reached, and, seldom used cached data will be replaced with essential data. For simple cache consistency management, instead of prefetching file blocks from the back-end servers, our approach is to prefetch whole files from the back-end servers, found in [Flin02]. Servers in the virtual back-end can be one of these types; high performance computers, or workstation clusters, secondary data storages. Service-oriented grid architectures, such as OGSA, virtualize back-end servers as “services”. The components of the back-end servers are as follows:

- Bootstrap Nodes: The bootstrap nodes keep super peer lists for each community in the virtual laboratory. After a participant node instantiates the MVM, the node first finds the super peer for the community, by looking up the lists in the bootstrap nodes. In case the first super peer in the list is not available, the node then finds the second super peer in the list. This process will be continued to the end of the list, if the node cannot find the right super peer for its community. Peer groups in the community periodically select their super peer and update the lists in the bootstrap nodes.
- Information Service Servers: The information service servers maintain the resource lists and contact details for each node, so that the super peer can discover resources by using them. Whenever the virtual laboratory users connect to the super peer, it first registers its resource information to the information service server. The super peer will aggregate the nodes according to the resource information retrieved from the information service servers. The directory servers will be used for our information service servers to provide a well defined syntax for objects with distinguished names.
- Image Servers: The image servers allow the MVMs to be instantiated anywhere in the community, and to be migrated on-demand. The image servers archive the static

MVM states, including the *OS-level VM* and *Queue VM* image in the MVM. The images in our virtual laboratory are virtualized images for dedicated raw machines, in that they provide additional abstraction layers for the existing systems.

- Data and Application Servers: The data servers provide the participant nodes with user data and applications for the virtual laboratory. The user data includes information that is required to operate specific experiments. The application servers provide the participant nodes with the executable applications for the MVM. The application may be model itself, if the VM decoder supports for translating the model into executable file. In case the executable UML technology [Luz04] or MDA process are incorporated into our *VM decoder*, we envision experimenting with a wide range of models in the modeling phase. In this situation, our queuing mechanism and the model executable engine (VM decoder) should be interfaced and connected by the *VM interface controller* with *VM interface logics*, and the network connections should be controlled by the connectivity component in the MVM. There are several on-going projects related to the model executable virtual machine [Balc03] and they are works in progress.
- Authentication and Authorization Servers: Authentication servers issue, revoke, and manage PKI-based certificates for virtual communities. Authorization servers issue, revoke, and manage role attribute certificates and policy attribute certificates. We adopt the Role-based Access Control (RBAC) [Sand96] method using attribute certificates for our virtual laboratory, as it facilitates managing and enforcing authorization in large-scale scalable systems. The Policy Management Points (PMPs), Policy Decision Points (PDPs), and the policy repositories are located in the virtual backend for above scheme.
- Component Servers: The component servers provide the *Components VM* with the reconfigurable components. The component servers act as component repositories for a wide variety of configuration components for MVMs. In the modeling phase, we do not define the specific component model for the specific platform. The “platform independent component model” [Ziad02] will be mapped to the “platform dependent

component model”. However, the component servers should provide the runtime reconfigurability for the MVMs, so they should maintain the platform dependent components for MVMs. Thus, we should store the platform dependent components in the component repositories, after we finish translating the “platform independent component model” to the “platform dependent component model”. Although the components are constrained to the specific environment, they are considered as logical, as they give only logical configurations for the specific experiment. It is up to the control processing element to determine and map these logical configurations.

4.1.2 Virtual Front-Ends

The virtual front-end is the super peer module of our virtual laboratory. We use the term “virtual front-end” rather than “front-end”, in that the virtual front-end node is itself a MVM node with additional flexibility, reconfigurability, and dynamicity.

When an input packet arrives at the virtual front-end, the first step is to process a header by decapsulating the input packet. After the header processing is complete, the jobs and data are enqueued. Then the virtual front-end node sets policies by using the reconfigurable components (see Figure 12). These policies include the scheduling, connectivity management, and sharing rules policies. We enforce the policies for the virtual cluster at the broker module of the virtual front-end node. As we stated in Section 3.1, the broker module is configured by our *Components VM* in the MVM. Thus, the policies that are enforced by the *Components VM* can be reused, or shared by disseminating the components to the nodes in the virtual cluster.

The main role of the virtual front-end is to schedule the incoming jobs in the *Queue VM* to the available resources in the virtual cluster. The scheduling problems in parallel computing, such as the “process-to-processor mapping problem”, are known to be “NP-complete problems” in its general form [Traf02]. The design of the local and global scheduler is beyond the scope of this thesis. However, by using our *Components VM*, we can configure scheduling policies to our schedulers in the virtual front-end nodes, such as First Come First Served, Select-Least-Loaded, and Select-Fixed-Sequence [Buyy99], giving flexibility to the local and global schedulers. For a specific experiment in the virtual community, the virtual front-end controls the connectivity and sharing rules for

the virtual cluster. The connectivity and sharing rules consist of the logical components, and are manipulated by the *Components VM* in the MVM. Thus, we should map the logical connectivity and sharing rules to the physical connectivity and sharing rules for the virtual cluster. We will discuss the “connectivity component” and their mapping procedure in Section 4.2.1 and Section 4.2.2, respectively. We introduce the partitioning strategy in Section 4.2.2, in order to avoid of congestion and to partition the virtual community in an efficient way. In summary, the *Queue VM* in the super peer’s MVM will act as the virtual front-end for the virtual cluster, and the components of *Components VM* will be retrieved from the component servers in the virtual backend for the specific experiment. Finally, the *Components VM* in the front-end will map these logical components to the actual job situations for the virtual cluster.

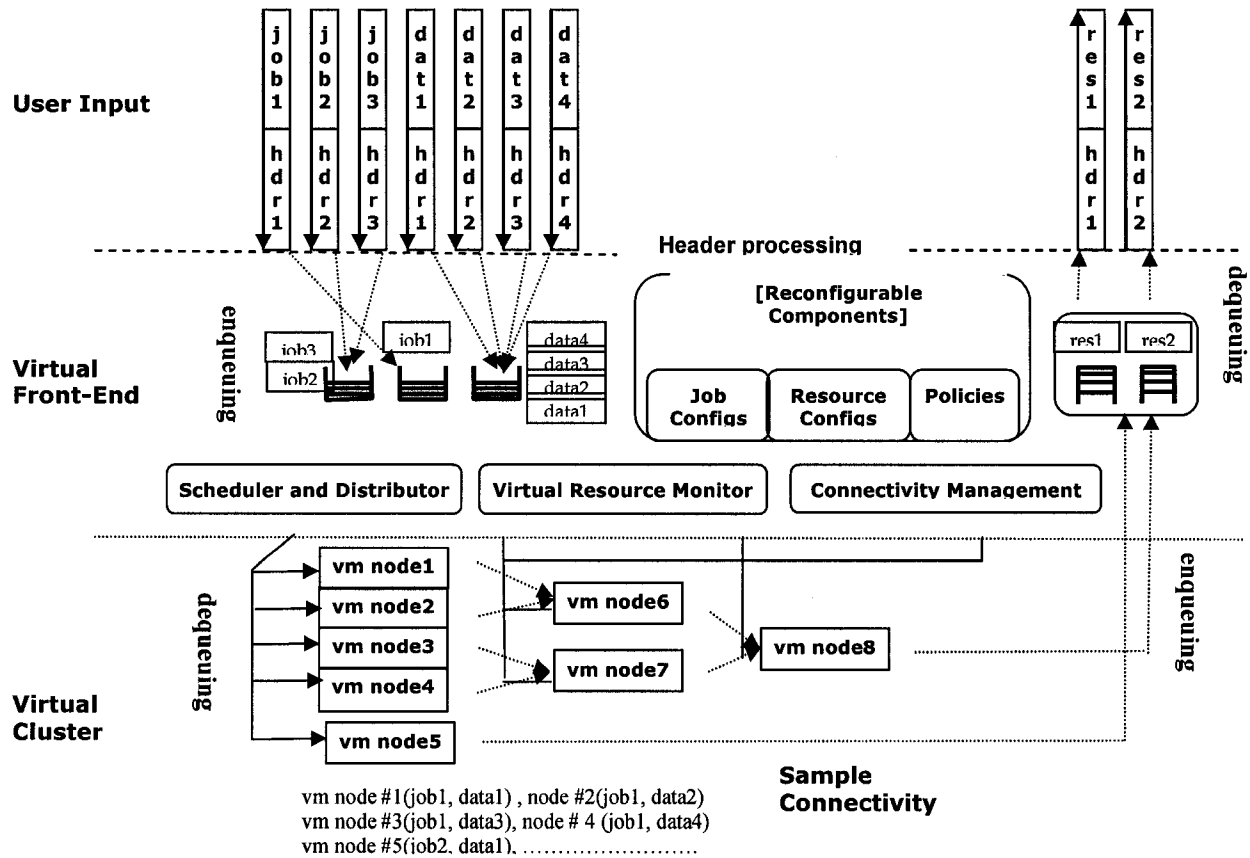


Figure 12 Virtual Front-End and Virtual Cluster

4.1.3 Virtual Clusters

The virtual cluster in our virtual laboratory is dynamically and on-demand organized as a group of MVMs. The virtual front-end node initially retrieves the available resource information of the virtual cluster from the information service server. Then the virtual front-end node notifies a group of available nodes in the community to organize the virtual cluster. The notified nodes respond by trying to connect to the virtual front-end node. The virtual front-end node instantiates the virtual switch module and then allocates its ports to the nodes. When the appropriate “MVM connect request” arrives to the virtual switch, then a new port can be allocated for the MVM by the virtual switch. This enables a physical connection to be established between the MVM node and the virtual switch in the virtual front-end node. The virtual switch polls the arrival of data from each node and then manipulates the forwarding or dropping of incoming requests. Each node is then identified by the port number of the virtual switch, and will be further organized by the connectivity rules. The “logical connectivity” component describes the connectivity of the virtual cluster by the logical node id numbers. The node id numbers can be replaced with the port numbers and the virtual switch numbers. The virtual switch sets the maximum number of ports. However, within the port ranges, it is not required to allocate ports in advance. When every successful “incoming request” arrives to the virtual switch, a new port is allocated until the maximum port number is reached. The virtual switch can also be organized in a hierarchical way. Consider that five hundred nodes join the specific experiment, and the maximum number of ports is twenty. Suppose further, that we adapt the simple partitioning scheme, such as one virtual switch for each twenty nodes. We might create twenty virtual switches, in addition to the higher level virtual switch, for twenty virtual switches. The remaining nodes might be used for redundancy purposes. In this situation, the higher level virtual switch allocates its ports for only virtual switches, and each virtual switch acts as a port for the high level virtual switch. Although the dynamic, virtual parallel computer (see Figure 7) is recursively organized, the recursive hierarchy should be hidden to the experimenter’s perspective. The experimenter initially sends a job to only a single virtual front-end (super peer), and the selected virtual front-end then gives some capable nodes to the super peer roles if

required. The initial virtual switch in the front-end can be at the highest hierarchy level. We can connect the virtual switch and the node element by linking both the *Queue VM*'s in the MVMs. Thus, at the highest level, we see the control processing element equipped with a virtual switch as well as a virtual cluster. At the next highest level, we still see the control processing element equipped with a virtual switch, and a virtual cluster. This recursive view can be maintained until the virtual cluster is required to split. The control processing elements at the higher level are only concerned with the control processing elements at the next level, and cascade the updates to next level by allowing each VM node to reconfigure their components. The Beowulf-class systems [Salm98] also showed how to organize the cluster system by using the tree of switches. However, virtual clusters with virtual switches are used for our system to accommodate geographically distributed computing resources and users. Thus, the latency between the virtual switches and general processing elements should be considered in this situation. We might set the range of latency for a virtual switch or reconstruct the virtual cluster periodically, allowing only nodes within the range to be connected to the virtual switch.

In case one of ports dies from the heavy load or failure, the virtual switch deallocates the port, and notifies another candidate node in the community. Then, the virtual switch allocates the new port for the candidate node. During this procedure, the virtual switch maintains a table for each port, and its corresponding physical address. For the pipelined applications, the virtual switch also maintains stage information for nodes in the table, activating the appropriate nodes at each stage. The results for each stage are routed to the nodes for the next stage by the virtual switch. However, in some cases, it is desirable to have the direct connections between the nodes in the virtual cluster, avoiding the message routing overhead in the virtual switch. In this situation, according to the connectivity configuration, the virtual switch commands the nodes for the direct connections, such that the queue VM in each MVM connects to each other. The problems we have to deal with are how we authenticate and authorize between nodes in the virtual community. We will briefly discuss these issues in Section 4.3.

4.2 Virtualized Networks

4.2.1 Connectivity component

When we run certain kinds of parallel applications, we need to have a virtual topology in order to specify the logical arrangement of tasks. In some situations, we are also required to specify each stage in the virtual topology for high performance pipelined computing², such as the *butterfly computation* of the *Fast Fourier Transform* (FFT) [Kent01]. However, in current parallel applications, we need to denote the virtual topology at the program level. Table 2 shows the method of usage of the Message Passing Interface (MPI) program model, for the virtual topology specified in Figure 13(a).

Table 2: A usage of virtual topology in a MPI program for Figure 13(a)

```
1  /* new communicator */
2  MPI_Comm graphcomm;
3
4  /* number of nodes in the virtual topology */
5  int num_of_nodes = 7;
6
7  /* Array of integers describing node degrees. The ith entry is the total number of
8  neighbors of the first i graph nodes */
9  int index[7] = {1,2,3,4,7,10,12};
10
11 /* Array of integers describing graph edges */
12 int edges[12] = {5,5,6,6,1,2,7,3,4,7,5,6};
13
14 /* Whether or not the rank reordering is allowed */
15 int mapping = 0;
16
17 /* New communicator with virtual topology (graph)*/
18 MPI_Graph_Create (MPI_WORLD_COMM, num_of_nodes, index, edges, 19
mapping, graphcomm);
```

² Parallel operations in this paper are denoted as “coarse grained parallel operations” rather than “fine grained parallel operations”. “Task parallel” operations are categorized in “coarse grained parallel operations” while “instruction level parallel” and “loop-level parallel” operations are categorized in “fine grained parallel operations”.

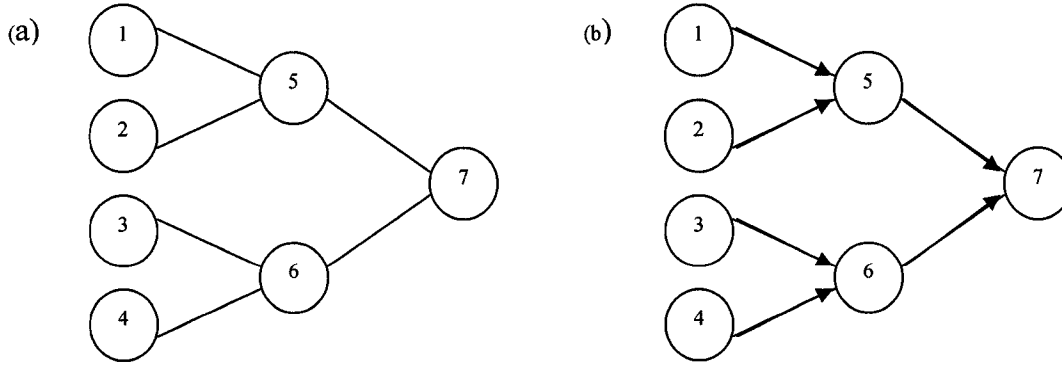


Figure 13 Simple Virtual Topology

Consider the situation, where we have modified the virtual topology specified in Figure 13(a) by replacing it with another virtual topology, which uses the same number of vertices in the same program. In current parallel system designs, for instance, MPI, we have to rewrite and recompile the program, to test another virtual topology model in the same program. Consider also, the situation where we use the Directed Acyclic Graph (DAG) model, to specify each stage of computation. We also need an additional programming effort to specify each stage in the MPI program. Thus, whenever we modify the virtual topology or stages of computations in basically the same program, we have to rewrite and recompile the MPI programs.

In Section 3.1, we stated that every MVM node has light-weight reconfigurable components that determine the intrinsic behaviour of each node. Meanwhile, MVM nodes in the virtual front-end determine the essential behaviours of the virtual cluster. The “connectivity component” only belongs to the MVM nodes in the virtual front-end. Each user in the virtual laboratory is able to edit or create a new “connectivity component”, and then upload it to component storage in the virtual back-end. Otherwise, they are able to simply reuse the existing “connectivity component”. Once we run a parallel job with a “connectivity component” (see Figure 14), we do not need to recompile it, whenever we modify a virtual topology, or stage information in basically the same program. The component-based software technology allows this “reconfigurable” functionality [Wegd01]. Our *Components VM* adopts the component-based software technology, in order to provide the MVM with the runtime

reconfigurability. Figure 14 describes a simplified view of the “connectivity component”. It has a “connectivity profile” as an input, and five external interfaces: *retrieve*, *update*, *add*, *delete*, and *store*. To denote the virtual topology in Figure 13, the “connectivity profile” includes the fields corresponding to lines 5, 9, and 12 of the above MPI program in Table 2, along with the type of virtual topology and the stage information for each node. We do not specifically define the file format for “connectivity profile”. However, it should follow the standard file format, and should be well structured to support efficient add, delete, query and update the virtual topology data. The *retrieve* interface is used to retrieve current virtual topology information. This information is utilized when the *Components VM* maps the process topology (virtual topology) into the processor topology. The *update* interface allows us to reconfigure virtual topology, including the stage information. The *add* interface and *delete* interface are used to add new virtual topology, and delete old virtual topology, respectively. The *store* interface allows us to save the added and modified virtual topology, and then generates another “connectivity profile”. The newly generated “connectivity profile” is then transferred to the experimenter’s node and/or the virtual backend for later use. The *Components VM*, outside the running parallel program, dynamically applies virtual topologies, along with their stage information to the virtual cluster. By using our virtual parallel computer, a wide range of traditional coarse grained parallel operations can be performed in a similar way to fine grained parallel operations, since the parallel procedure is transparent at the program level. We often find that parallel program debugging is much more difficult, than serial program debugging. Thus, we envision our virtual parallel computer as a means to lessen debugging difficulties, for the sub domain of parallel programming.

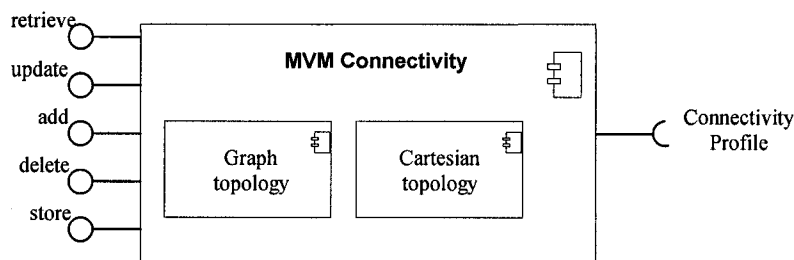


Figure 14 Connectivity Component

After the virtual topology information is retrieved by the *Components VM*, the *Components VM* then conducts the “virtual topology” to the “processor topology” through our mapping procedure, for the virtual cluster.

4.2.2 Partitioning and mapping procedure

As we mentioned earlier in Section 4.1, the bootstrap nodes maintain the super peer lists, for the virtual community. The super peer lists are kept on a regional basis in the bootstrap nodes. When an experimenter participates in a certain experiment, his/her MVM node downloads the super peer list of his/her region from the bootstrap node. It contacts the super peers in the list consecutively, and measures the delay between each super peer node and itself. Then, it attempts to connect the super peer that has the least delay from the node. The MVM node does not connect to the super peer until it receives the “ACK” message from the super peer. When the super peer reaches its maximum capacity, it responds to the MVM node by a “NAK” message. If the MVM node receives the “NAK” message from a super peer, it attempts to connect the super peer that has the next minimum delay from the MVM node. This procedure continues until the MVM node is accepted by the super peer that responds to the MVM node by the “ACK” message. The super peer partitions the nodes in the virtual cluster, once it has reached its maximum number of nodes considering its capacity. In this situation, the highest level of a super peer selects the next highest level of super peers, and keeps a hierarchical tree for each partition, where each edge in the hierarchical tree denotes the communication cost. Whenever the higher level super peer finishes the partitioning of the virtual cluster and selects the next higher level super peers, the higher level super peer updates the super peer list in the bootstrap nodes by removing itself and adding the next level super peers (see Table 3). In the super peer list shown in Table 3, the super peer that conducts the partitioning (*ip address: 137.231.54.69*) is replaced with the newly selected super peers (*ip address: 137.231.54.42, 137.231.54.77, 137.220.231.57, 137.220.231.65*) after the partitioning. If the next level super peer’s maximum capacity is “*a*” number of nodes, and the “*b*” number of nodes are allocated for the partitioning, it still has $a - b$ of nodes, where $a > b$, are available for new nodes. Thus, after a partitioning takes place, the newly participating nodes attempt to contact the newly generated super peers. By using the

virtual switch modules along with the above method, we emulate the hierarchical tree of switches, which has been widely adopted for clustering systems, such as Beowulf-class [Salm98].

Table 3 : The updated super peer list in the bootstrap node

(Note : This list is an artificial for illustration)

Super peer list (Before Partitioning)	Super peer list (After Partitioning)
Windsor, ON, CA : 137.207.120.219 137.207.101.155 (137.231.54.69) 137.232.33.45	Windsor, ON, CA 137.207.120.219 137.207.101.155 * (137.231.54.69) (137.231.54.42) (137.231.54.77) (137.220.231.57) (137.220.231.65) 137.232.33.45 * denotes "not available" for new connections

The maximum hierarchical level of the tree is subject to the maximum setting value of the routing metric (hop count) of the peer group, avoiding a certain peer group to be expanded indefinitely. Thus, the super peer that has already reached at the maximum level in the hierarchical tree does not partition its virtual cluster, and simply sends the "NAK" message for the new connection attempts, if it already reached the full capacity. Using this method, we avoid the saturation of the super peer due to the large number of nodes in the virtual cluster. The other reason for partitioning the virtual cluster is to efficiently map "virtual topology" to "processor topology". We divide the virtual cluster into the "affinity groups", in order to minimize the communication costs inside each group, once the mapping procedure is fulfilled. The *min-sum clustering algorithm* [Vega03] and *k-medoids clustering algorithm* [Kull04] can be applied for the above purpose. The *min-sum clustering problem* [Vega03] is defined as,

"Consider a set V of n points endowed with a distance function $\delta : V \times V \rightarrow R$. These points have to be partitioned into a fixed number k of disjoint subsets C_1, C_2, \dots, C_k so

as to minimize the cost of the partition, which is defined to be the sum over all clusters of the pairwise distance in a cluster. ”

To find an optimal solution for the above problem turns out to be NP-hard, however, the polynomial time approximation algorithms are presented in [Vega03]. The *k-medoids* clustering algorithm using *k-medoids* heuristic is widely used because of its simplicity [Kull04]. This algorithm starts with a set of arbitrary k nodes that has been chosen as an initial solution. Then, this solution is iteratively improved as described in [Kull04]. In each iteration, the input nodes are partitioned into k sets, by associating each node with its closest node in the current solution; The *k-medoids* of the sets in the partitions are returned as the output of each iteration. The *k-medoid* is acquired by recomputing each cluster center (*medoid*) and has the highest “network affinity” in the current partition. The “network affinity” metric was introduced in [Hata98], allowing us to measure the affinity of a certain node to other nodes in a partition. The “network affinity” of node i in the partition is defined as,

$$a_i = (\sum_{j=1}^{|P|} D_{i,j})^{-1} \quad (1)$$

where $D_{i,j}$ is the distance from node i to node j and $|P|$ denotes the number of the available nodes in the partition. Let us denote the number of iterations by l . Then the worst case complexity of computing *k-medoids* is $O(n^2l)$ and each iteration requires $O(nk)$ time for the *k-medoids* heuristic [Kull04]. The second phase takes into account only the clusters that have more nodes than their predefined partition capacities. The lower valued “network affinity” nodes in the cluster are reassigned to the other closest clusters in which the predefined capacity has not been reached, so that the capacity constraint is met in the end. The second phase iterations also require $O(nk)$ time for the worst case complexity.

Each super peer at each level, keeps track of the total number of the nodes in its rooting subtree, and updates it for every partitioning. The weight of an inner node vertex (super peer) in the hierarchical tree denotes the total number of nodes in its rooting subtree. The

multi-level mapping using the tree architecture is presented in [Hata98]. Our job distribution scheme is similar to the *multi-level* mapping. For each mapping procedure, at first, all processes are assigned to the super peer at a certain level in the hierarchical tree. The appropriate level is determined by the privilege of a user, and it will be described in Section 4.3. Then, it distributes the processes to the nodes at the next higher level, by referencing the weights of the nodes. A node with a higher weight is assigned first. The total count number of processes is subtracted when each distribution to the inner node is complete. This procedure is recursively applied to the tree in a breadth first manner until the total count is zero. Thus, all processes are placed at the lowest level super peer(s) in the hierarchical tree in the end.

Once the job distribution is complete to each partition, we have to select “good nodes” for mapping inside the partition. In the initial phase of mapping, we might first select the nodes inside “module”s, according to the “module” size and its performance, where the “module” [Hata98] is a group of processing elements that has a high communication path between them, such as Massive Parallel Processing (MPP), and Symmetric Multiprocessing (SMP) machines. However, in a dynamic environment where the heavy workload is assigned and varied, we need to distribute the workload efficiently to the available processing elements, while taking into account the performance for each parallel application. Our selection of “good nodes” for mapping inside the partition is as follows:

Let $A_i(t_j)$ be a normalized value of “network affinity” a_i at time instant t_j , and $C_i(t_j)$ be the fraction³ of the computational power of the processing element that is available to a task at time instant t_j . Our formula to measure the quality of node for mapping inside the partition is defined as

$$W_i(\lambda, t_j) = \lambda A_i(t_j) + (1 - \lambda) C_i(t_j) \quad (2)$$

where $0 \leq \lambda \leq 1$, $t_j > 0$, $j = 0, 1, 2, \dots$ and A_i , C_i lies in the interval $[0, 1]$.

³ The measurement of C_i is discussed in [Subh99].

In above formula, t_j denotes the time instant for each measurement, and the measurement delay is ignored. The average quality of nodes for mapping in the partition k at time instant t_j is defined as

$$Q_k(\lambda, t_j) = \frac{\sum_{i=1}^{|P_k(t_j)|} W_i(\lambda, t_j)}{|P_k(t_j)|} \quad (3)$$

where $|P_k(t_j)|$ denotes the number of nodes in the partition k at time instant t_j , $t_j > 0$, $j = 0, 1, 2, \dots$, and $0 \leq \lambda \leq 1$.

According to the above formula, we can give a “weight relationship” between the “network affinity” and the available “computational power”. In case we run a “communication-bound” parallel application, we can give a higher λ . Conversely, if we run a “computation-bound” parallel application, we can give a lower λ . Note that, when $\lambda = 1$, we ignore the available “computational power”, and select the nodes according to the “network affinity”. When $\lambda = 0$, we ignore the “network affinity”, and select the nodes according to the available “computational power”. In our criteria, the “good node” for our mapping procedure is located in a dense region, and high “computational power” is available. In contrast, the “bad node” for mapping is located in a sparse region, and low “computational power” is available. The highest W_i at the first time instant of the partition is chosen as an initial super peer. We assign the nodes in the virtual topology into the nodes in the virtual cluster, by using the W_i value in above formula, where the node with the higher W_i value is assigned first. Although the W_i value of node i at a time instant t_{j-1} is high, the node might be measured as a low W_i value at a time instant t_j . This situation happens when the node suffers from a heavy load or network congestion, acquiring the low priority, for the next mapping procedure as a result.

4.3 Policy-based Reconfigurable Components

According to Appleby [Appl04], “policy-based computing” is “a software paradigm that incorporates a set of decision-making technologies into its management components in order to simplify and automate the administration of computer systems”. The policy components in the MVM are designed to achieve the above viewpoint, allowing dynamic adjustment of the behaviour of the group of MVMs at runtime without modifying its internal implementation. The policies in our virtual laboratory framework are deployed and reconfigured at different levels of abstraction. We divide the policies into three levels: inter-domain level, domain (virtual community) level, and general node level. The inter-domain level policies may require a mediator to manage semantic heterogeneity and integration of multiple heterogeneous policies for each domain. We mandate that inter-domain resource sharing in the virtual laboratory be subject to the inter-domain policies. Figure 15 depicts how the inter-domain policies are generated.

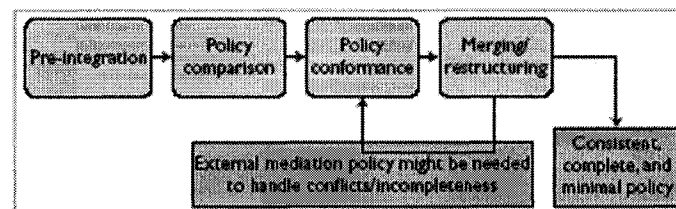


Figure 15 Policy-integration for multi-domain environments [Josh04, pp:50]

The domain level policies apply to the virtual community to specify the security rules, resource sharing rules, privileges for each participant, fault recovery mechanisms, scheduling and monitoring mechanisms, and so forth. The domain-level policies include how and when the partitioning takes place, and how to monitor each partition. For both inter-domain and domain level policies, the Policy Management Point (PMP), Policy Decision Point (PDP), and policy repository should be located in the virtual back-ends. Our “policy-based reconfigurable components” are located in the Policy Enforcement Point (PEP) that is the broker module of the highest level super peer in the hierarchical tree. Once the highest level super peer in the virtual community enforces these policies,

the lower level super peers in the hierarchical tree retrieve them from the highest level super peer if needed.

The general node level policies determine the interface logics, such as encoding/decoding types, queue type, maximum number of *Queue VMs*, and so forth. They also specify how to monitor the performance and availability of each node, and what metrics are used for them. The general node level policies do not require external PDP and PMP, thereby allowing the user to set his/her policies for MVM.

The “Policy-based Reconfigurable Components” consist of three main building blocks: security handlings, fault tolerance, and scheduling/monitoring.

The security handlings deal with both authentication and authorization. The authentication policy specifies what kind of authentication mechanism is used for a certain virtual community. It also includes whether the credential repository service is used, and if used, how the credentials can be retrieved from the credential repository service. For scalability, we advocate the PKI-based GSI authentication model [Fost02], which supports “single sign on” and “delegation” capabilities. However, the dynamic policy reconfiguration is not necessarily required for authentication in the current phase of our virtual laboratory. Meanwhile, the authorization policy determines whether a certain user is allowed to do an action by using a certain amount of resources. The Role-based Access Control (RBAC) [Sand96] is emerging as an authorization mechanism for large-scale systems in which both policies and user roles are stored in attribute certificates to provide integrity. It is based on user-to-role assignment and role-to-permission assignment, thereby avoiding the complexity of traditional access control mechanism and reducing management cost as a result [Zhou04]. Each user in a virtual community is assigned to an appropriate role in accordance with his/her resource contribution to the virtual community. When a user sends a job with job profile to the broker module of its super peer, the super peer determines, based on the user’s role information, whether it should manipulate or reject the request or forward it to the higher level super peer in the hierarchical tree. After receiving a job with a job profile from the user, the super peer decapsulates the job profile that includes resource requirement for the specific job. It then queries to the PDP, whether the resource requirement for the user is

legitimate. The PDP retrieves role ACs, policy ACs, and X.509 PKCs from the policy and credential repository or the user's file system, then checks whether the user and his/her role is valid and whether the resource requirement is legitimate for his/her role. If the request is permitted, PDP sends an "ACK" message to the super peer (PEP), allowing the super peer to schedule or forward the submitted job from the user. Otherwise, the super peer simply rejects and returns the request from a user with a warning message. [Zhau04] describes the detailed steps of RBAC-based authorization.

The context and content-based constraints for the extended RBAC Model is discussed in [Josh04]. In our virtual laboratory, the PEP, which can be reconfigurable by using our *components VM*, provides different interfaces to the PDP that can be invoked depending on those constraints, such as the time, system status (failure, congestion, etc), collaborating entities, roles, and so forth. By the above method, a user can use more resource than the predefined resource usage limit for his/her role if a virtual cluster is idle. The user has to use less resource if a virtual cluster is congested, and vice versa. Resource sharing can also be dynamically applied between different virtual communities. We might mandate that resources in a virtual community be accessible to other virtual communities only by previous reservation and/or low congestion in a virtual community. The automatic runtime policy reconfiguration and enforcement allows our virtual machines to be self-configurable. It also allows us to envision the self-configuring, self-protecting, and self-management capabilities of those found in "autonomic computing" [Gane03] vision.

The monitoring reconfigurable policy determines how often the resources in a virtual community are monitored, and what metrics are used to monitor them. Each monitoring result is reported to the PEP, and then the PEP determines what policy components should be replaced to optimize the virtual cluster. In the test phase, we might generate some signal events to simulate fault and congestion situations, and then test how the appropriate actions can be taken (policy components reconfigurations, failure recovery, etc). The scheduling reconfigurable policy determines what policy will be applied for resource scheduling in a virtual community. The optimal scheduling policy depends on each monitoring result, allowing the scheduling policy to be dynamically adjusted for

systems status in the virtual community. The scheduling policy is also affected by the resource reservation policy. The resource reservation policy determines whether the reservation is allowed, how the reservation is made, and what requirements need to be satisfied for resource reservation.

The fault tolerance reconfigurable components consist of MVM checkpointing and MVM failure recovery policy components. The MVM checkpointing policy determines what kind of checkpointing method is used for a virtual community and how often the checkpointing is to be conducted. As the MVM is based on OS-level VM technology, OS-level VM based checkpointing can be adopted for our virtual laboratory. The strength of OS-level VM based checkpointing is that all volatile execution states of running processes (including disks, memory, CPU registers, I/O devices, etc) can be capsulized [Sapu02]. Thus, the whole running MVM, including *Queue VMs* and *Components VMs*, and its applications can be capsulized. Suppose that we run a parallel program using a cluster of nodes, which requires a month of computation. We can store *capsules* for every node to a storage system on a daily basis. Only the data blocks with different hash values would be transferred for each *capsule* update to reduce the amount data to be transferred. In case one of the nodes has crashed due to a hardware failure and cannot be recovered, the last checkpointed capsule of the crashed node in the storage system can be instantiated to another available node. Other nodes do not have to start a program again from the beginning due to a failure of a crashed node. They are able to roll back to the previous checkpointed location, and resume parallel operations.

5 IMPLEMENTATION AND PERFORMANCE ANALYSIS

5.1 Overview of MVM toolkit

We have developed the Multi-layer Virtual Machine (MVM) toolkit v.0.1 to evaluate the essential functionality of the MVM framework. This toolkit is a testing tool for MVM framework in distributed systems and grids and it is a work in progress. The current features of MVM toolkit v.0.1 are as follows:

- Source-code level parallel job distribution: Traditional parallel computing architectures, such as MPI and PVM, need to (re)compile parallel jobs for every participating node in deploy-time. MVM toolkit provides parallel job distribution at the source-code level, allowing us to (re)compile and instantiate parallel jobs in runtime for participating nodes.
- Component-based parallel job (re)configuration: In the MVM framework, each user sends a job profile to a resource broker to request a unique runtime environment for each usage case. The MVM toolkit encodes/decodes the plain-text job profile data structure into “Simple Object Access Protocol” (SOAP) encoding/decoding format and vice versa, intended to provide resource sharing on heterogeneous environment by using standard web services.
- Virtual network approach for parallel job execution: When we run certain kinds of parallel applications, we need to have a virtual topology in order to specify the logical arrangement of tasks. In current parallel applications, we need to denote the virtual topology at the programming level.

Message Passing API for MPI

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);  
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status);
```

In MPI, we have to specify the source/destination field at the source-code level to send and receive messages for parallel computing.

Message Passing API for MVM toolkit

```
int enqueueVM( void *arg, char *size, int data_type, JobInfo *jobinfo );  
int dequeueVM( void *arg, char *size, int data_type, JobInfo *jobinfo );
```

Note that there is no source/destination field for message passing API in the MVM toolkit. Each user can create a virtual network in runtime, specifying the virtual topology for the user's parallel job. All connectivity information is determined outside the parallel program, allowing us to provide efficiency and runtime-reconfigurability for parallel job execution. Additionally, this scheme allows grid resource scheduler or allocator to select and map process-to-processor in heterogeneous environments in a dynamical way, as the message passing API for MVM is not bound to specific source or destination job id at the source-code level.

- P2P Web services and P2P socket approach: Each node has both server and client module for socket and web services. Each node can be both a resource provider and consumer. Each node publishes its service by using Web Service Description Language (WSDL), and accesses other nodes by using a Simple Object Access Protocol (SOAP) interface. Each node also has a socket server and a client module. The socket server module acts as a communication gateway for the MVM, while the socket client module is used for sending various requests to other nodes including a resource broker. Each socket server process is transient in that it only exists while the MVM parallel tasks are running.
- On-demand creation and termination: The MVM processes do not have to run all of the time. Whenever a node is invoked from other nodes, it can initialize itself and launch tasks for a particular use. The “on-demand” creation and termination mechanism for the MVM is as follows:

1. Only the Apache server runs for each node and the MVM process is not loaded yet.



Figure 16 Initial phase (No MVM process loaded)

2. After contacting a bootstrap node, the MVM client retrieves the broker address and invokes the components of the broker node by using SOAP. The MVM client sends a job profile data structure to the broker node at this phase.

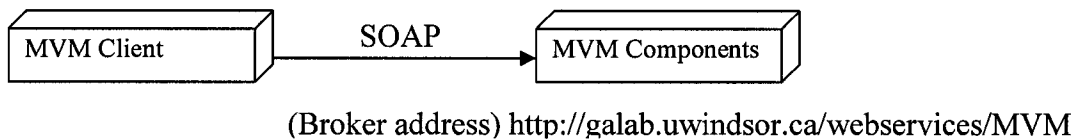


Figure 17 MVM toolkit starts its operation by SOAP invocation

3. The MVM process is loaded by the SOAP invocation, and the process instantiates the MVM proxy and the MVM queue threads, if required.

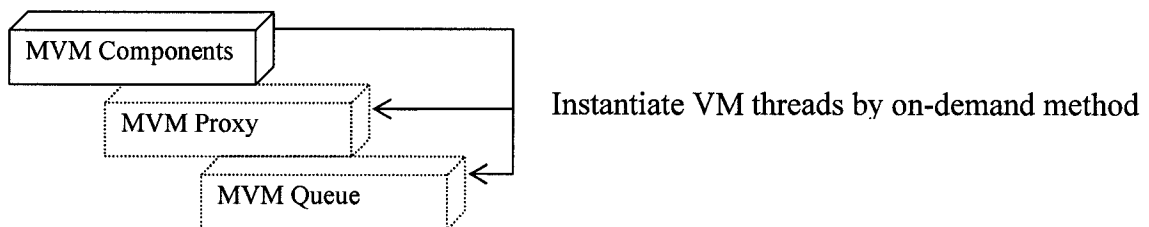


Figure 18 Spawns VM threads by on-demand method

4. The broker node selects a job group and awakes all the nodes in that job group by using the above method. Each node in a job group instantiates its proxy and queue VM module if required.

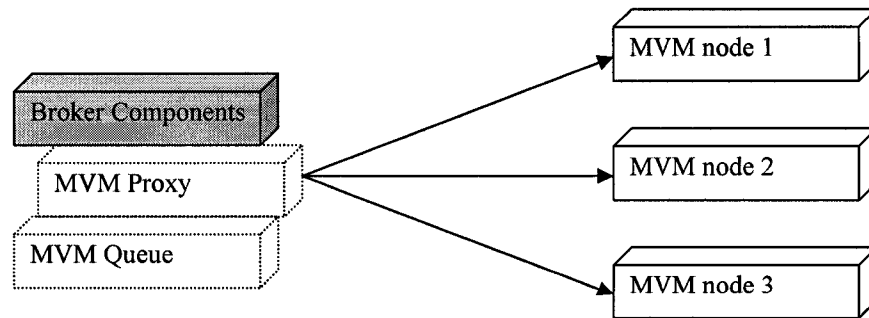


Figure 19 Awakes other nodes in a job group by SOAP invocation

5. Each node now can communicate with other nodes by using its proxy with BSD sockets.

6. The broker node generates the job instantiation messages for each participating node, and sends these messages to each participating node in a concurrent way. According to the job instantiation messages, queue VM spawns child processes and initialize the Inter Process Communication (IPC) channel for local processes. Processes in a local host enqueue or dequeue their data and instructions via the IPC channel, and processes between different hosts enqueue or dequeue their data via their proxies.

7. When a parallel job has finished its operation, it reports to a resource broker. The resource broker then broadcasts a job termination message to a job group. A proxy module for each job group reads the message, and sends a “SIGTERM” signal to all on-demand created processes. If no MVM job is running at this phase, it returns to the original phase.

5.2 Implementation and specification of the MVM toolkit

5.2.1 Implementation of the MVM toolkit

The MVM toolkit v.0.1.0 has been implemented in C/C++, and tested in Linux OS. The implementation and porting issues for different architectures are discussed in Appendix. A UML (Unified Modelling Language) class diagram that displays the core entities of the MVM toolkit and their associations is shown in Figure 20. The main entities of the MVM toolkit are:

1. **MVM Proxy:** The MVM Proxy is the main communication module for the MVM toolkit. It distributes and receive source job file, and compile the job. The proxy module of a broker (super peer) allocates resource and generates the job instantiation messages for job groups. When a job instantiation message has been received, it parses and passes the message to queue VM, allowing queue VM to spawn child processes and initialize the IPC channel. It also works as a communication gateway between different nodes, as it multiplexes, enqueues, and dequeues the job messages for child processes of each node.
2. **MVM Components:** The MVM Components is the SOAP implementation of an abstract job profile module of the MVM toolkit. The job profile includes the resource requirements, such as the number of nodes required, and the virtual network of these nodes. The virtual network is mapped to the physical network according to the *components VM* of a broker. The MVM Components of a broker works as an Object Request Broker (ORB). In grid environments, heterogeneous resources provide a wide variety of services implemented by different languages and different platforms. We aim to interoperate between heterogeneous environments by using web services, allowing a job profile data structure to be sent and received by remote object invocation in a transparent way on heterogeneous platforms.
3. **MVM Connectivity:** The MVM Connectivity describes the logical arrangement of tasks. It also describes the stages of tasks, whereby each task can activate its operation in a certain stage.

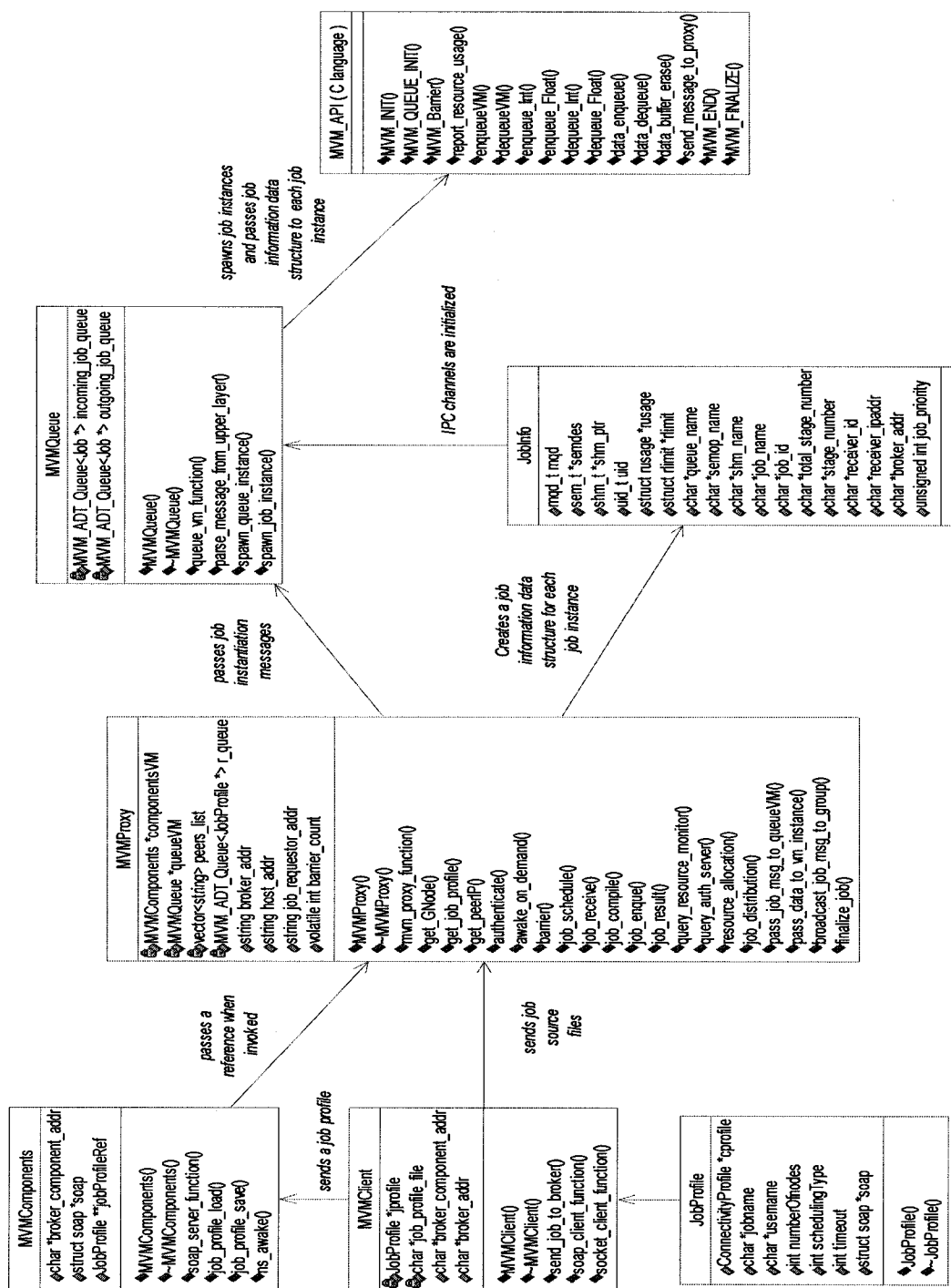


Figure 20 UML Diagram of core MVM toolkit modules

We envision to reuse a wide variety of parallel programming patterns by specifying the connectivity profile, allowing a user to store and update his/her connectivity profiles and experiment with different parallel communication patterns without direct modification of parallel programs at the source-code level.

4. MVM Queue: The queue VM of the MVM toolkit enqueues and dequeues the job instantiation messages from the upper layer, spawns child processes and initializes IPC channel for tasks. While MVM Proxy manipulates socket communications, the MVM Queue manipulates the IPC between its child processes. The device driver level, generic data communication interface by queue VM is planning to be integrated into a future version of the MVM toolkit.
5. MVM API library: The MVM API library provides an interface between MVM applications and the MVM kernel. The *MVM_INIT(JobInfo *jobinfo, char **argv)* function initializes MVM application's data structure. Whenever a queueVM spawns child processes, it passes the data structure to each process. The data structure contains the job name, job id, IP address of a process, connectivity information, and the broker address. The data structure is generated by resource allocator module for each child process. The *MVM_INIT(JobInfo *jobinfo, char **argv)* function stores this data structure from the MVM kernel into the application's data structure, allowing each child process to send or receive messages according to this data structure. The *enqueueVM()* function invokes a virtual machine module, and the virtual machine module packs a data and enqueues the data to a destination node as determined by the connectivity profile and resource scheduler. The *dequeueVM()* function unpacks and dequeues data, stores data into a buffer, and returns a source job id. Note that the *dequeueVM()* function does not have to wait for a message from a particular source job id. Once the task of a child process has been completed, it calls the *MVM_END(JobInfo *jobinfo)* function and waits for the "SIGTERM" signal. When it receives the "SIGTERM" signal from the kernel, it deallocates its application data structure and clears its IPC channel.

6. **MVM Client:** The MVM Client module first contacts the MVM bootstrap node, and retrieves a resource broker (super peer) address. After retrieving the resource broker address, it connects to the resource broker and then sends a data structure of job profile to the resource broker. The resource broker determines whether or not the resource request of the client is legitimate, and then receives source files of a job from the client. The runtime compilation of the MVM applications is a default option of the MVM toolkit.

5.2.2 MVM toolkit v.0.1.0 protocol description

The MVM toolkit v.0.1.0 Protocol is a textual, message-oriented protocol consisting of Command and Response messages exchanged between MVM nodes. The protocol is line-based and the end of a line is represented by the new line token that is a line-feed character. The Command messages have the following format.

Synopsis

COMMAND P1 P2...Pn

Description

COMMAND is the name of the command to be executed by MVM node(s), followed by zero or more required parameters (P1...Pn). Individual parameters are separated by one or more delimiter character, whereby parameter names are case-insensitive. The **COMMAND** consists of version, command value, and command code, separated by delimiter character. Table 4 describes the command messages supported by the current version of MVM toolkit.

Example: MVM/0.1 500 JOB_RESULT:MVM_INT:2*2 1 2 3 4

Table 4: MVM toolkit v.0.1.0 Command Messages

Command Code	Value	Parameters
QUERY_BROKER_ADDR	100	YES

AUTH_REQUEST ⁴	200	YES
SCHED_REQUEST	300	YES
JOB_SEND	400	YES
JOB_RESULT	500	YES
FINALIZE_JOB	550	YES
DIST_JOB	600	YES
INSTANTIATING_JOB	700	YES
IPC_DATA	800	YES
ENQUEUE_DATA	900	YES
BARRIER_REQUEST	950	YES

The response messages are sent from the target to the source to report the status of command execution.

Synopsis

RESPONSE

Description

RESPONSE message comes in two forms: a positive response, and a negative response. Both positive and negative responses carry no parameters. Negative responses have minus value for status code. The **RESPONSE** consists of version, status value, and status code. Table 5 and Table 6 describe the response messages supported by the current version of MVM toolkit.

Example 1: MVM/0.1 504 RECV_JOB_SUCCESS

Example 2: MVM/0.1 -700 BARRIER_FAILURE

Table 5: MVM toolkit v.0.1.0 Response Messages (SUCCESS)

Status Code (Success)	Value	Parameters
AUTH_REQUEST_SUCCESS	501	N/A
RECV_JOB_SUCCESS	502	N/A
SCHED_REQUEST_SUCCESS	503	N/A
QUEUE_JOB_SUCCESS	504	N/A
INSTANTIATING_JOB_READY	505	N/A
COMPLETED_ENQUEUE	506	N/A

⁴ Authentication/Authorization has not yet implemented in the current version of MVM toolkit.

BARRIER_SUCCESS	507	N/A
-----------------	-----	-----

Table 6: MVM toolkit v.0.1.0 Response Messages (FAILURE)

Status Code (Failure)	Value	Parameters
QUERY_BROKER_FAILURE	-100	N/A
AUTH_FAILURE	-200	N/A
SCHED_REQUEST_FAILURE	-300	N/A
JOB_RECV_FAILURE	-400	N/A
QUEUE_JOB_FAILURE	-500	N/A
JOB_COMPILE_FAILURE	-600	N/A
BARRIER_FAILUER	-700	N/A

The following example shows the typical sequence of the submission, distribution, and instantiation of a job involved in MVM toolkit.

- MVM node to MVM Broker
 - > *MVM/0.1 400 JOB_SEND (parameters)*
 - < *MVM/0.1 502 RECV_JOB_SUCCESS*
 - > *MVM/0.1 300 SCHED_REQUEST (parameters)*
 - < *MVM/0.1503 SCHED_REQUEST_SUCCESS*
- MVM Broker to all participating nodes in a job group
 - > *MVM/0.1 600 DIST_JOB (parameters)*
 - < *MVM/0.1 504 QUEUE_JOB_SUCCESS*
 - > *MVM/0.1 700 INSTANTIATING_JOB (parameters)*
 - < *MVM/0.1 505 INSTANTIATING_JOB_READY*
 - < *MVM/0.1 950 BARRIER_REQUEST (parameters) // synchronization for starting a job*
 - > *MVM/0.1 507 BARRIER_SUCCESS // in a job group*

The MVM Queue module parses the parameters of *INSTANTIATING_JOB* command, spawns VM instances, passes the parameters to each VM instance, and binds each VM instance with message queues. The format of *INSTANTIATING_JOB* command is:

Syntax

MVM/0.1 700 INSTANTIATING_JOB:(job name):(job id):(IP address):(total stage number):(stage number):(receiver id list):(IP addresses of receiver ids):(broker IP address)

Table 7: Parameters of *INSTANTIATING_JOB* Command

<i>Parameter</i>	<i>Description</i>
<i>Job Name</i>	<i>The name of a job</i>
<i>Job ID</i>	<i>Each VM instance has a unique job id allocated by resource scheduler</i>
<i>IP address</i>	<i>The host machine's IP address of a VM instance</i>
<i>Total stage number</i>	<i>Total stage number specified by the connectivity profile</i>
<i>Stage number</i>	<i>Stage number of a VM instance</i>
<i>Receiver ID list</i>	<i>The job ID list of receivers specified by the connectivity profile</i>
<i>IP addresses of receiver Ids</i>	<i>The IP addresses of receiver Ids</i>
<i>Broker IP address</i>	<i>The super peer (resource broker)'s IP address</i>

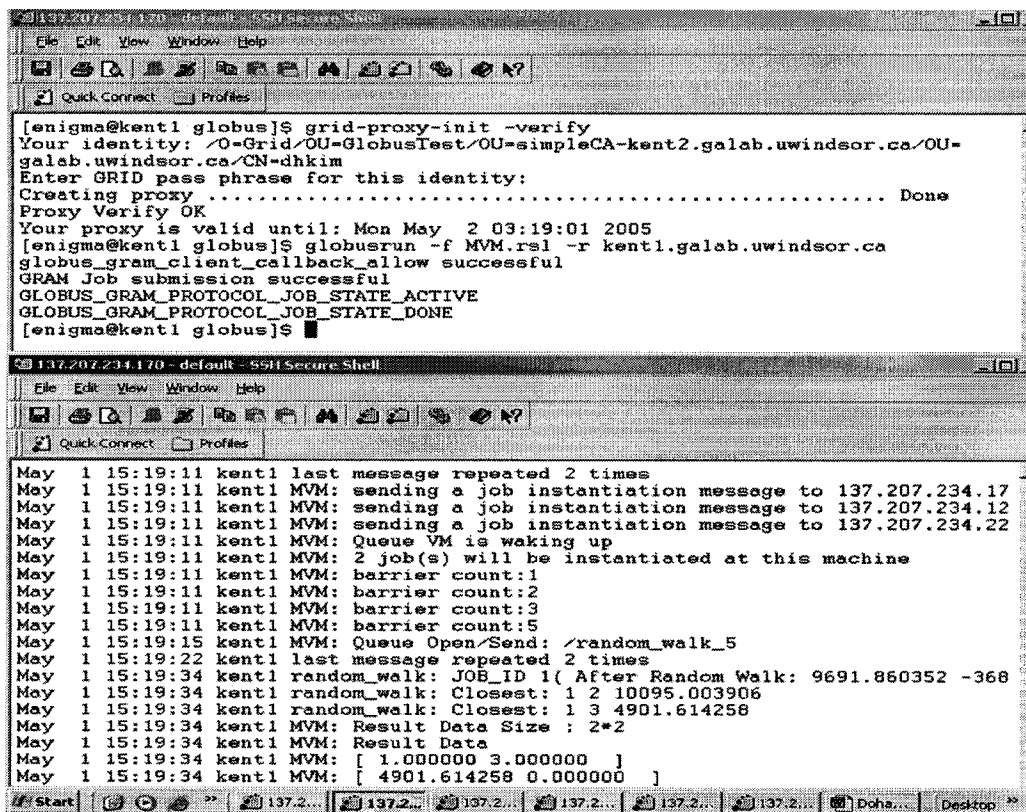
5.3 Experimental results and analysis

Our approach advocates a “divide and conquer” approach to verifying the design described in previous chapters. As our design takes a multi-layered approach, we could test each individual layer with its components and then test interface for each layer. The multi-layered design approach allows us to reduce the complexities of implementing and testing for the overall design. After we have tested each layer, we test several applications with different job and connectivity profiles. Experiments have been performed with several applications of the MVM toolkit v.0.1.0 on Linux-based systems. Our experimental setup consisted of five nodes: three nodes including a broker node are located in the computer science laboratory in the University of Windsor, and two nodes including a bootstrap node are located in the home network connected to a cable modem with 100-Mbps Ethernet. We used the Apache web server 2.0.52, gSOAP 2.7.0⁵, Globus 3.2.1, POSIX IPC, POSIX Thread, and BSD Socket, running under Fedora Core 3 Linux

⁵ gSOAP is an Open Source SOAP toolkit for C/C++ and it is available at <http://www.cs.fsu.edu/~engelen/soap.html>.

systems (kernel 2.6.9) for our implementation and experiment. The broker node runs on an Intel Pentium IV 2.8 GHz with 512Mbytes of RAM, and the bootstrap node runs on an Intel Celeron 2.4 GHz with 128Mbytes of RAM. Three other nodes run on AMD SEMPRON 2400+ (1.7GHz) with 256Mbytes of RAM, Intel Pentium III 800MHz with 256Mbytes of RAM, and Intel Pentium IV 1.4GHz with 512Mbytes of RAM, respectively.

In our test scenario, for a given number of iterations, four nodes of the MVM run a “random walk” sample program in a parallel way. One node waits for four walkers, collects the final location of each walker, and then calculates the shortest distance among walkers. Figure 21 shows the capture screen of the MVM toolkit running on Globus 3.2.1. It is not obligatory to run MVM toolkit on Globus tool. However, we do aim to take advantage of some security and resource management features of the Globus toolkit.



```
[enigma@kent1 globus]$ grid-proxy-init -verify
Your identity: /O=Grid/OU=GlobusTest/OU=simpleCA-kent2.galab.uwindsor.ca/OU=
galab.uwindsor.ca/CN=dhkim
Enter GRID pass phrase for this identity:
Creating proxy ..... Done
Proxy Verify OK
Your proxy is valid until: Mon May  2 03:19:01 2005
[enigma@kent1 globus]$ globusrun -f MVM.rsl -r kent1.galab.uwindsor.ca
globus_gram_client_callback_allow successful
GRAM Job submission successful
GLOBUS_GRAM_PROTOCOL_JOB_STATE_ACTIVE
GLOBUS_GRAM_PROTOCOL_JOB_STATE_DONE
[enigma@kent1 globus]$
```

```
May 1 15:19:11 kent1 last message repeated 2 times
May 1 15:19:11 kent1 MVM: sending a job instantiation message to 137.207.234.17
May 1 15:19:11 kent1 MVM: sending a job instantiation message to 137.207.234.12
May 1 15:19:11 kent1 MVM: sending a job instantiation message to 137.207.234.22
May 1 15:19:11 kent1 MVM: Queue VM is waking up
May 1 15:19:11 kent1 MVM: 2 job(s) will be instantiated at this machine
May 1 15:19:11 kent1 MVM: barrier count:1
May 1 15:19:11 kent1 MVM: barrier count:2
May 1 15:19:11 kent1 MVM: barrier count:3
May 1 15:19:11 kent1 MVM: barrier count:5
May 1 15:19:15 kent1 MVM: Queue Open/Send: /random_walk_5
May 1 15:19:22 kent1 last message repeated 2 times
May 1 15:19:34 kent1 random_walk: JOB_ID 1( After Random Walk: 9691.860352 -368
May 1 15:19:34 kent1 random_walk: Closest: 1 2 10095.003906
May 1 15:19:34 kent1 random_walk: Closest: 1 3 4901.614258
May 1 15:19:34 kent1 MVM: Result Data Size : 2*2
May 1 15:19:34 kent1 MVM: Result Data
May 1 15:19:34 kent1 MVM: [ 1.000000 3.000000 ]
May 1 15:19:34 kent1 MVM: [ 4901.614258 0.000000 ]
```

Figure 21 Running MVM toolkit on Globus 3.2.1

Figure 22 shows the results of “random walk” sample program for different number iterations with three and five nodes of MVMs. As shown by Figure 22, five nodes obtain better results for the large number of iterations, while three nodes are more efficient for the small number of iterations.

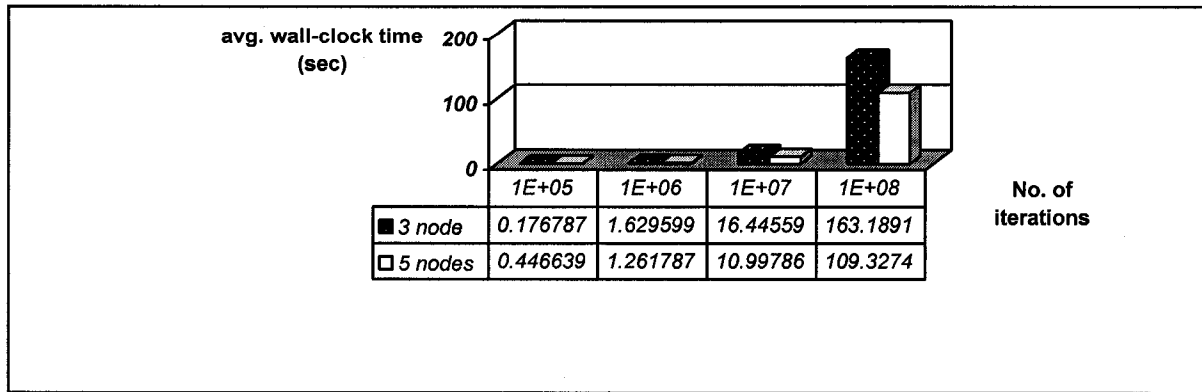


Figure 22 Elapsed time for 3 and 5 MVM nodes for the “random walk” program

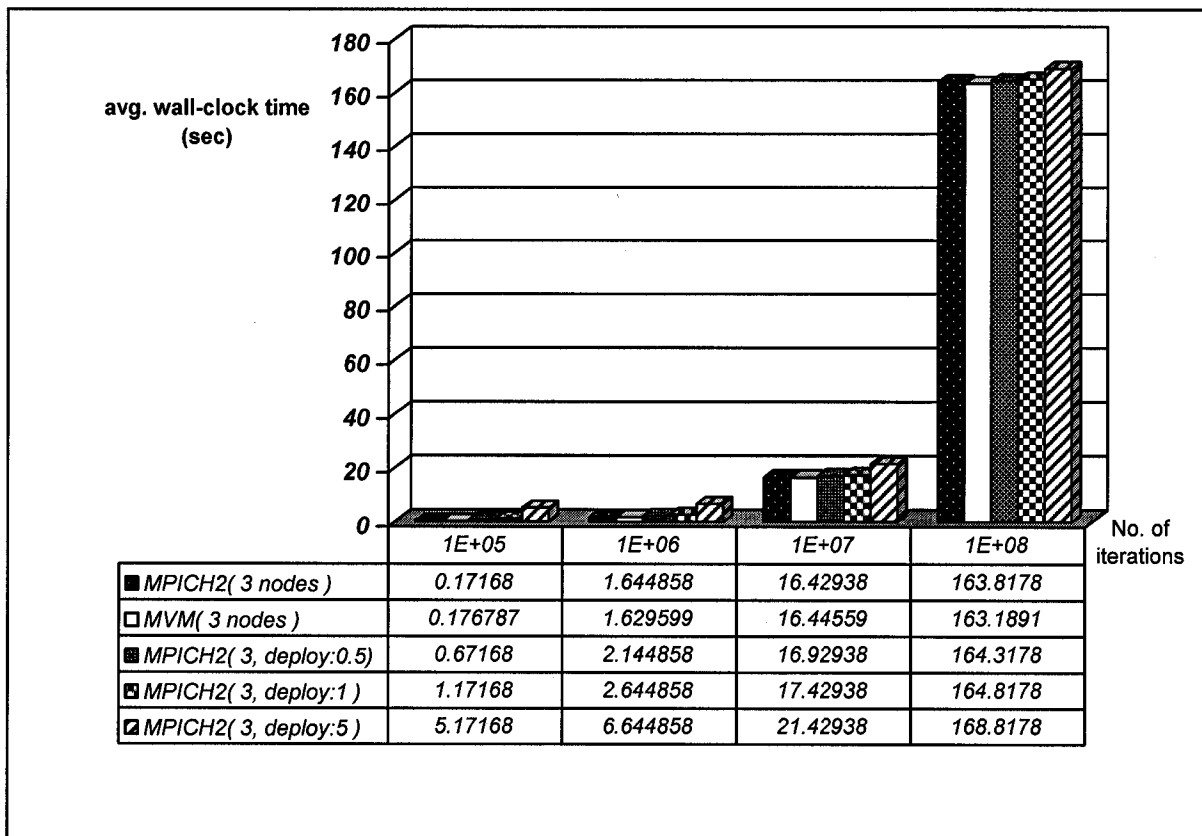


Figure 23 Elapsed time for random walk program (MPICH2 & MVM)

The main performance bottleneck of the small number of iterations is the message passing overhead. Since our experiment has been conducted with the same number of processes for both three and five nodes, the computations with five nodes contain more message-passing between different hosts than that of three nodes.

Figure 23 shows the performance comparison between MVM and MPICH2 v.1.0.1⁶ deploy field in the table in Figure 23 is our assumed total time gaps between the automatic runtime deployment of a MVM job and the manual deployment of a MPICH2 job. The first row in the table in Figure 23 shows the ideal case for MPICH2 in which the human intervention time for deployment is ignored. The rest of them include the assumed time gaps between the automatic runtime deployment and the manual deployment for a parallel job.

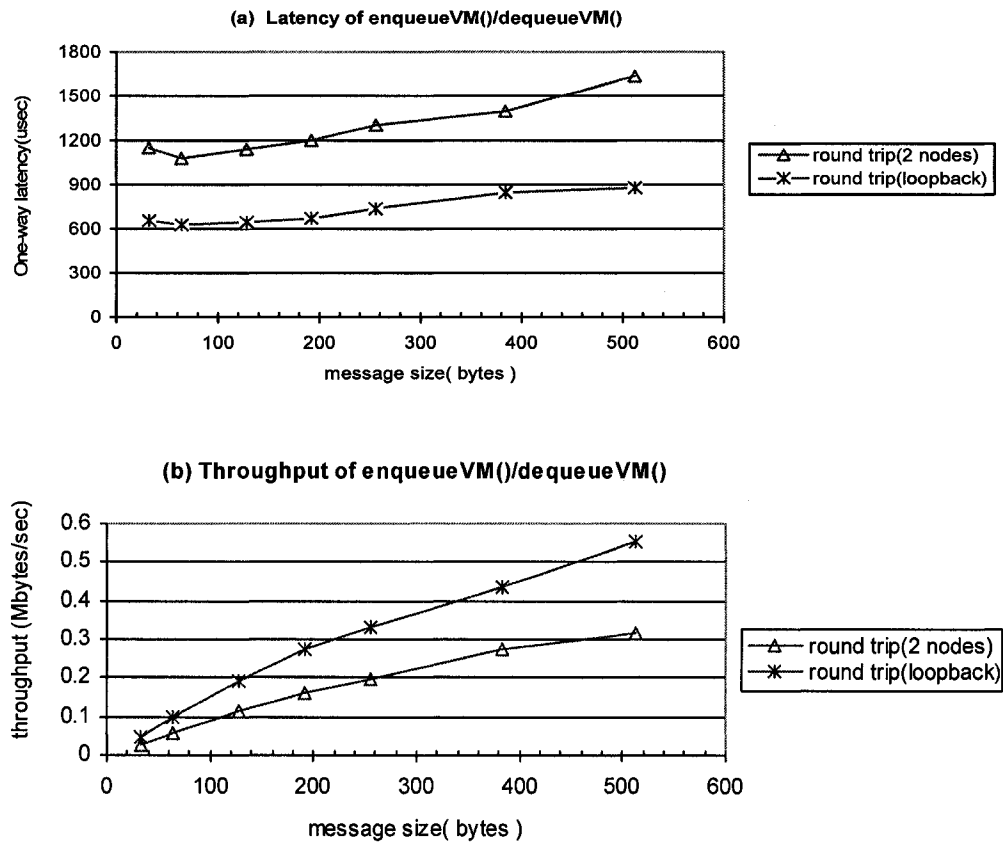


Figure 24 Latency and throughput with different message sizes

⁶ MPICH2 is an implementation of the Message-Passing Interface (MPI). MPICH2 is 1.0.1 was released on March 2, 2005. It is available at <http://www-ux.mcs.anl.gov/mpi/mpich2/>

As the number of iterations increases, five nodes of the computations gain more performance benefit than that of the three nodes by achieving parallelism with different hosts. The message passing latency and the throughput are important factors affecting the MVM performance for communication bound parallel applications. Figure 24 shows the latency and throughput of different message sizes for the message passing by use of the `enqueueVM()` and `dequeueVM()`. We benchmark the latency of `enqueueVM()` and `dequeueVM()` by measuring the half of the round trip delay of the different sizes of messages. According to the latency values, the throughput values have been derived. The throughput values grow as the message sizes increase. Besides other critical performance factors, such as CPU, memory, network hardware and its protocol, operating systems, applications' algorithm, and so on, the performance of `enqueueVM()` and `dequeueVM()` heavily rely on the message passing protocol and the VM proxy overhead. As our tool takes an initial step towards the development of the MVM framework in grid environments, there remains much work to do in order to achieve moderate performance while providing essential features that are described in previous chapters.

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

Whenever we virtualize something in a computer system, we can often reduce management cost by providing a predefined interface for virtualization. For instance, if we virtualize the network, the network management cost can be lessened. We have described how the virtualization technology can provide runtime flexibility and automaticity for grids. By specifying the virtual topology and network information outside the parallel programming source codes rather than inside the programming source codes, we could reuse parallel communication patterns by reusing virtual topologies. We further provide parallel job distribution at the source-code level, allowing us to (re)compile and instantiate parallel jobs in runtime rather than in deploy time. We have demonstrated that our connectivity mechanism for parallel job execution, along with parallel job distribution at the source-code level, can provide the runtime flexibility for the virtual laboratory.

In grid environments, heterogeneous resources provide a wide variety of services implemented by different languages and different platforms. We aim to interoperate between heterogeneous environments by using our components VMs, allowing a job profile data structure to be sent, received, and scheduled by remote component invocation in a transparent way on heterogeneous platforms. We also have justified the reconfigurability of our virtual machine. It allows each user to request and (re)configure a runtime environment, and test various research models for reduced cost and time.

Although the experiment has been conducted with a small set of nodes, the MVM enhances its performance with a bigger number of nodes and data sizes. The most valuable experimental result is its efficiency. We could reuse a same parallel communication pattern for different applications. We also could distribute and deploy parallel jobs for distributed memory machines in an efficient manner. Suppose that we modify values in an application source file and run it on multiple machines, or change some configuration values. In traditional parallel computing architectures, such as MPI

and PVM, we have to connect each machine, update each programming source file, recompile or reconfigure them for each modification. In the same scenario, MVM allows a user to modify his/her job profile or programming source file, and then simply reconfigure a group of machines in runtime in an automatic manner. In our framework, the virtual machines can also be “on-demand” created and destroyed. This capability can give a further flexibility and automaticity in grid environments. It lessens human efforts and intervention for configuring and deploying parallel jobs, reducing management and maintenance cost as a result. These runtime functionalities of MVM allow us to move one step closer to the autonomic computing vision described in previous chapters.

6.2 Future work

As our implementation is in its initial phase towards building the MVM framework, we have not yet implemented all the features described in previous chapters. Our flexible “reconfigurability” mechanism of MVM has not yet implemented in a policy-based manner. We plan to enhance our “reconfigurability” mechanism into a policy-based mechanism. We envision that the automatic runtime policy reconfiguration and enforcement allows our virtual machines to be self-configurable at a certain level. The implementation and testing of interface module for existing OS-level VMs also has to be done to further verify our framework. The main issue of source-code level job distribution is a security. A malicious or faulty code can harm entire system. We aim to take advantage of the OS-shielding capability of OS-level VMs. In addition, we plan to include the code verifier at a resource broker, avoiding the distribution of faulty code to a job group. We also plan to enhance the MVM communication performance by implementing generic data communication interface for our Queue VM. Finally, we plan to extend our test for Multiple Program Multiple Data (MPMD) model by which virtual laboratory users can distribute and run multiple different parallel source-code level jobs to available nodes via resource brokers in an automatic manner, and then apply a wide variety of parallel communication patterns in runtime.

Bibliography

- [1]. [Alle01] G. Allen, T. Damlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, B. Toonen, "Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus", *Proc. Of 2001 ACM/IEEE Conference on Supercomputing*, pp:52-52, 2001.

- [2]. [App104] K. Appleby, S.B.Calo, J.R. Giles, K.W. Lee, "Policy-based automated provisioning", *IBM Systems Journal*, vol. 43, num. 1, utility computing, webpage: <http://www.research.ibm.com/journal/sj/431/appleby.html>, 2004.

- [3]. [Balc03] M.J. Balcer, "An Executable UML Virtual Machine", ModelCompilers.com, webpage: http://www.omg.org/news/meetings/workshops/UML%202003%20Manual/02-3_Balcer.pdf, 2003.

- [4]. [Bavi04] A. Bavier, L. Peterson, M. Wawrzoniak, S. Karlin, T. Spalink, T. Roscoe, D. Culler, B. Chun, M. Bowman, "Operating System Support for Planetary-Scale Network Services, *Proc. Of the 1st USENIX/ACM Symposium on Networked Systems Design and Implementation*, pp:253-266, 2004.

- [5]. [Barh03] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, A. Warfield, "Xen and the Art of Virtualization", *Proc. Of the 19th ACM Symposium on Operating Systems Principles*, pp:164-177, 2003.

- [6]. [Bilk02] A.Bilke, O. Klischat, E.U. Kriegel, R. Rosenmuller, "Component-based software development - a practitioner's view", *SDPS Journal of Design & Process science*, vol.6, Issue: 4, pp:52-62, 2002.

- [7]. [Buch02] K. Buchacker, V. Sieh, H. Hoxer, "Implementing a User Mode Linux with Minimal Changes from Original Kernel", *9th International Linux System Technology Conference*, pp: 72-82, 2002.

- [8]. [Bugn97] E. Bugnion, S. Devine, M. Rosenblum, "Disco: running commodity operating systems on scalable multiprocessors", *Proc. Of the Sixteenth ACM Symposium on Operating Systems Principles*, pp:143-156 , 1997.

- [9]. [Buyy99] R. Buyya, "High Performance Cluster Computing vol.1", *Prentice Hall, ISBN 0-13-013784-7, pp: 111,503,505, 1999.*
- [10]. [Camp99a] A. Campbell, H. D. Meet, M. Kounavis, K. Miki, J. Vicente, D. Villela, "A Survey of Programmable Networks", *ACM Computer Communications Review*, vol. 29, Issue: 2, pp: 7-24, 1999.
- [11]. [Camp99b] A. Campbell, J. Vicente, D. Villela, "Virtuosity: Performing Virtual Network Resource Management", *7th IEEE/IFIP International Workshop on Quality of Service (IWQOS'99)*, pp: 65-76, 1999.
- [12]. [Camp99c] A. T. Campbell, M. E. Kounavis, D. A. Villela, J. Vicente K. Miki, H. G. De Meer, K. S. Kalaichelvan, "Spawning Networks", *IEEE Network Magazine* vol. 13, Issue: 4, pp: 16-30, 1999.
- [13]. [Casa95] J. Casas, D. L. Clark, R. Conuru, S. W. Otto, R. M. Prouty, J. Walpole, "MPVM: A Migration Transparent Version of PVM", *Computing Systems*, vol. 8, Issue: 2, pp:171-216, 1995.
- [14]. [Chen01] P.M. Chen, B.D. Noble, "When virtual is better than real", *Proc. Of 8th Workshop on Hot Topics in Operating Systems*, pp: 133-138, 2001.
- [15]. [Crea81] R.J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development*, vol. 25, Issue: 5, pp: 483-490, 1981.
- [16]. [Dasw02] N. Daswani, H. Garcia-Molina, "Query-Flood DoS Attacks in Gnutella", *ACM CCS*, pp: 181 -192, 2002.
- [17]. [Dike01] J. Dike, "User-mode Linux", *Proc. Of the 5th Annual Linux Showcase and Conference*, pp:3-14, 2001.
- [18]. [Estr01] F. Estrella, Z. Kovacs, R. McClatchey, "Model and Information Abstraction for Description-Driven Systems", *CHEP '01 conference*, webpage : <http://www.ihep.ac.cn/~chep01/paper/8-053.pdf>, 2001.
- [19]. [Ensi03] Ensim, "Ensim Virtual Private Servers", webpage : http://www.ensim.com/products/materials/datasheet_vps_051003.pdf, 2003.

- [20]. [Figu03] R. Figueiredo, P. Dinda, and J. Fortes, "A Case for Grid Computing on Virtual Machines", *Proc. Of IEEE ICDCS 2003*, pp: 550-559, 2003.
- [21]. [Flin02] J. Flinn, S. Sinnamohideen, N. Tolia, M. Satyanarayanan, "Data Staging on Untrusted Surrogates", *Proc. of the 2nd USENIX Conference on File and Storage Technologies*, webpage: <http://citeseer.ist.psu.edu/flinn02data.html>, 2002.
- [22]. [Flyn66] M. Flynn, "Very High-Speed Computing Systems", *Proc. of the IEEE*, vol. 54, Issue:12, pp:1901-1909, 1966.
- [23]. [Fost01] I. Foster, C.Kesselman, S.Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International J. Supercomputer Applications*, vol.15, Issue:3, pp:200-222, 2001.
- [24]. [Fost02] I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration", *Open Grid Service Infrastructure WG, Global Grid Forum*, webpage: <http://www.globus.org/research/papers/ogsa.pdf>, 2002.
- [25]. [Fost96] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit", *International Journal of Supercomputer Applications and High Performance Computing*, vol.11, Issue:2, pp:115-128, 1996.
- [26]. [Gane03] A. Ganek, "Autonomic computing: implementing the vision", *Autonomic Computing Workshop*, pp:1-1, 2003.
- [27]. [Garf03] T. Garfinkel, M. Rosenblum, "A Virtual Machine Introspection-based Architecture for Intrusion Detection", *10th Annual Network and Distributed System Security Symposium, NDSS' 2003*, webpage: <http://www.isoc.org/isoc/conferences/ndss/03/proceedings/papers/13.pdf>, 2003.
- [28]. [Gold74] R.P. Goldberg, G.J. Popek, "Formal Requirements for Virtualizable Third Generation Architectures" *In Communications of the ACM*, vol. 17, Issue: 7, pp: 412-421, 1974.
- [29]. [Grim97] A.S. Grimshaw, W.A. Wulf, and the Legion team, "The Legion vision of a worldwide virtual computer", *Magazine*

of *Communications of the ACM*, vol. 40, Issue:1, pp:39-45, 1997.

- [30]. [Hand03] K.A. Fraser, S. M.Hand, T. L.Harris, I. M. Leslie, I. A. Pratt, , “The Xenoserver computing infrastructure”, *Technical Report, University of Cambridge, Computer Laboratory, webpage: <http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-552.pdf>*, 2003.
- [31]. [Harr01] T. L. Harris, “Extensible Virtual Machines”, *PhD thesis, Churchill College, University of Cambridge, webpage: <http://www.cl.cam.ac.uk/~tlh20/papers/tim-harris-thesis-tr.ps.gz>*, 2001.
- [32]. [Hata98] T. Hatazaki, “Rank reordering strategy for MPI topology creation functions”, In 5th European PVM/MPI User’ s Group Meeting, volume 1497 of Lecture Notes in Computer Science, pp: 188-195, 1998.
- [33]. [Jian03a] X. Jiang, D. Xu, “SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms”, *IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pp:174-183, 2003.
- [34]. [Jian03b] X.Jiang, D. Xu, "VIOLIN: Virtual Internetworking on OverLay Infrastructure", *Department of Computer Sciences Technical Report CSD TR 03-027, Purdue University, webpage: <http://www.cs.purdue.edu/homes/dxu/pubs/violin.pdf>*, 2003.
- [35]. [Jian03c] X. Jiang and D. Xu, “vBET: a VM-Based Emulation Testbed”, *Proc. Of ACM SIGCOMM 2003 Workshops* , pp: 95-104, 2003.
- [36]. [Jian04] X. Jiang, D. Xu, R. Eigenmann, "Protection Mechanisms for Application Service Hosting Platforms", *to appear in Proc. Of IEEE/ACM Int'l Symposium on Cluster Computing and the Grid (CCGrid 2004), Chicago, IL, webpage: <http://www.cs.purdue.edu/homes/dxu/pubs/CCGrid04.pdf>*, 2004.
- [37]. [Josh04] Joshi, J.B.D. Bhatti, R. Bertino, E. Ghafoor, A., “Access-Control Language for Multidomain

Environments", *Internet Computing, IEEE*, vol.8, Issue : 6, pp: 40-50, 2004

- [38]. [Kamp00] P.H. Kamp, R. N. M. Watson, "Jails: Confining the Omnipotent Root", *Proc. Of the 2nd International SANE Conference*, webpage: <http://phk.freebsd.dk/pubs/sane2000-jail.pdf>, 2000.
- [39]. [Kent01] Kent, R.D., Majmudar, N., Schlesinger, M., "Distributing Fast Fourier Transform Algorithms for Grid Computing", *High Performance Computing Systems and Applications*, Kluwer Academic Publishers, Nikitas J. Dimopoulos (eds.), pp:407-426, 2001.
- [40]. [King02] S.T. King , G.W. Dunlap , S. Cinar, M. Basrai, P.M. Chen, "ReVirt: Enabling Intrusion Analysis through Virtual Machine Logging and Replay", *Proc. Of Symposium on Operating Systems Design and Implementation*, pp:211-224, 2002.
- [41]. [Klas04] Klasse Ojecten, *MDA information center*, webpage : <http://www.klasse.nl/english/mda/mda-introduction.html>, 2004.
- [42]. [Klep03] A. Kleppe, J. Warmer, W. Bast, "MDA Explained, The Model Driven Architecture: Practice and Promise", Addison Wesley, ISBN 0-321-19442-X, pp:16, 85, 2003.
- [43]. [Kozu02] M. Kozuch, M. Satyanarayanan, "Internet Suspend/Resume", *Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pp:40-46, 2002.
- [44]. [Krsu04] I. Krsul, A. Ganguly, J.Zhang, J.A.B. Fortes, R.Figueiredo, "VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing", *SC 2004*, webpage:<http://www.sc-conference.org/sc2004/schedule/pdfs/pap305.pdf>, 2004.
- [45]. [Kull04] M. Kull, "Fast Clustering in Metric Spaces", *University of Tartu, Faculty of Mathematics and Computer Science, MSc. Thesis*, webpage: http://www.egeen.ee/u/vilo/edu/Students/Meelis_Kull/Meelis_Kull_MSc_2.pdf, 2004.
- [46]. [Lind97] T. Lindholm, F. Yellin, "The Java Virtual Machine Specification", *The Java Series*, Addison-Wesley, Reading, MA, USA, January, 1997.

- [47]. [Litz92] M.J. Litzkow, M.Livny, M. W. Mutka, "Condor technical report", *Technical Report CS-TR-92-1069, University of Wisconsin*, webpage: <http://citeseer.ist.psu.edu/briker91condor.html>, 1992.
- [48]. [Luz04] M.P. Luz, A. R. Silva, "Executing UML Models", *WiSME 2004*, webpage: <http://www.metamodel.com/wisme-2004/present/20.pdf>, 2004.
- [49]. [Merw97a] J.E. van der Merwe, J.E., Rooney, S. Leslie, I.M. S.A. Crosby, "The Tempest - A Practical Framework for Network Programmability", *IEEE Network magazines*, vol.12, Issue:3, pp:20-28, 1997.
- [50]. [Merw97b] J. E. van der Merwe I. M. Leslie, "Switchlets and dynamic virtual ATM networks", *Proc. Of the Fifth IFIP/IEEE International Symposium on Integrated Network Management*, pp: 355-368, 1997.
- [51]. [Milo00] D.S. Milojicic, F. Dougkis, Y. Paindavein, R. Wheeler, S. Zhou, "Process Migration", *ACM Computing Surveys*, vol. 32, Issue:3, pp: 241-299, 2000.
- [52]. [Nytu02] J.P. Nyttun, "The UML Metamodel architecture", webpage: <http://fag.grm.hia.no/ikt2340/year2002/themes/executableUML/notes/Metamodel.pdf>, 2002.
- [53]. [OMG01] Object Management Group, "OMG Technology Explained", webpage: <http://www.omg.org>, 2001.
- [54]. [Osma02] S. Osman, D. Subhraveti, G. Su, J. Nieh, "The design and implementation of Zap: A system for migrating computing environments", *Proc. Of 5th USENIX Symposium on Operating Systems Design and Implementation*, pp: 361-376, 2002.
- [55]. [Pete02a] L.Peterson, T. Roscoe, "PlanetLab Phase 1: Transition to an Isolation Kernel", *PlanetLab Design Notes*, webpage: <http://www.planet-lab.org/pdn/pdn02-003.pdf>, 2002.
- [56]. [Pete02b] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", *Proc. Of ACM HotNets-I*, pp: 59-64, 2002.
- [57]. [Pren00] P.D. Preney, R.D. Kent, "Toward a Model of Models. Part

1", *High Performance Computing Systems and Applications*, Andrew Pollard et al, eds, Kluwer Academic Publisher, pp:33-38, 2000.

- [58]. [Pren99] P.D. Preney, "Towards a kernel architecture for modeling system", *MSc thesis, University of Windsor*, 1999.
- [59]. [Rajk98] R. Rajkumar, K. Juvva, A. Molano, S. Oikawa, "Resource Kernels: A Resource-Centric Approach to Real-Time Systems", *Proc. Of the SPIE/ACM Conference on Multimedia Computing and Networking*, pp:476-490, 1998.
- [60]. [Rich98] T. Richardson, Q. Stafford-Fraser, K. R. Wood, A. Hopper "Virtual Network Computing", *Magazines of IEEE Internet Computing*, vol. 2, Issue:1, pp:33-38, 1998.
- [61]. [Robi00] J.S. Robin, C.E. Irvine, "Analysis of the intel pentium's ability to support a secure virtual machine monitor", *Proc. Of the 9th USENIX Security Symposium*, webpage: http://www.usenix.org/publications/library/proceedings/sec2000/full_papers/robin/robin.pdf, 2000.
- [62]. [Rosh03] R. Roshandel, N.Medvidovic, "Modeling Multiple Aspects of Software Components", *SAVCBS ' 2003, Specification and verification Component-based System*, pp: 88-91, 2003.
- [63]. [Salm98] J. Salmon, C. Stein, T.L. Sterling. "Scaling of Beowulf-class Distributed Systems", *Proc. Of ACM/IEEE conference on Supercomputing (CDROM)*, pp:1-13, 1998
- [64]. [Sand96] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman, "Role based access control models", *IEEE Computer*, 29 February, pp:38-47, 1996.
- [65]. [Sapu02] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, M. Rosenblum, "Optimizing the Migration of Virtual Computers", *ACM SIGOPS Operating Systems Review*, vol. 36, Issue: SI, pp: 377-390, 2002.
- [66]. [Scha04] P. Schaumont, K. Sakiyama, A. Hodjat, I. Verbauwhede, "Embedded Software Integration of Coarse-grain Reconfigurable Systems", *18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, pp: 137-138, 2004.

- [67]. [Schm00] B. K. Schmidt, "Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches", *PhD thesis, Department of CS, Stanford University, webpage : <http://suif.stanford.edu/papers/schmidt00.ps.gz>*, 2000.
- [68]. [Shri76] B. D. Shriver , J. W. Anderson , L. J. Waguespack , D. M. Hyams , R. A. Bombet, "An implementation scheme for a virtual machine monitor to be realized on user microprogrammable minicomputers", *Proc. Of the annual conference*, pp: 226-232, 1976.
- [69]. [Smit01] J.E. Smith, "An overview of virtual machine architectures", *Technical Report, University of Wisconsin, webpage : http://swig.stanford.edu/~fox/cs241/readings/smith_vm_overview.pdf*, 2001.
- [70]. [Subh99] J. Subhlok, P. Lieu, Bruce B. Lowekamp, "Automatic Node Selection for High Performance Applications on Networks," *Proc. of the Seventh ACM SIGPLAN Symposium on the Principles and Practice of Parallel Programming (Atlanta, Georgia)*, pp: 163-172, ACM Press, 1999.
- [71]. [Suge01] J. Sugerman, G. Venkitachalam, B.H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor", *USENIX Annual Technical Conference*, pp: 1-14, USENIX Association, 2001.
- [72]. [Sund03] A. Sundararaj, P. Dinda, "Towards Virtual Networks for Virtual Machine Grid Computing", *Technical Report NWU- CS-03-27, Department of Computer Science, Northwestern University, webpages : <http://www.cs.northwestern.edu/~ais/nwu-cs-03-27.pdf>*, 2003.
- [73]. [Tane95] A.S. Tanenbaum, "Distributed Operating Systems", *Prentice Hall, Inc., ISBN 0-13-143934-0*, 1995.
- [74]. [Thom99] R. Thomas, "Mite: a basis for ubiquitous virtual machines", *PhD dissertation, University of Cambridge Computer Laboratory, webpage: <http://rrt.sc3d.org/download/research/mitethes.pdf>*, 1999.
- [75]. [Traf02] J.L. Träff, "Implementing the MPI Process Topology Mechanism", *Proc. of the 2002 ACM/IEEE conference on Supercomputing*, pp: 1-14, 2002.

- [76]. [Vega03] W. F. de la Vega, M. Karpinski, C. Kenyon, Y. Rabani, "Approximation Schemes for Clustering Problems". In *Proc. of the 35th Ann. ACM Symp. on Theory of Computing*, pp: 50–58, 2003.
- [77]. [Wald02] C. Waldspurger, "Memory Resource Management in VMware ESX Server", *Proc. Of the 5th symposium on Operating systems design and implementation*, pp:181-194, 2002.
- [78]. [Wegd01] Wegdam M. Almeida J.P.A., Pires L.F. and Van Sinderen M, "An approach to dynamic reconfiguration of distributed systems based on object middleware", *Proc. of 19th Brazilian Symposium on Computer Networks (SBRC 2001)*, webpage: <http://doc.utwente.nl/fid/1255>, 2001.
- [79]. [Whit02] A. Whitaker, M. Shaw, S.D. Gribble, "Scale and performance in the Denali isolation kernel", *ACM SIGOPS Operating Systems Review*, vol. 36, Issue: SI, pp: 195- 209, 2002.
- [80]. [Wiki] Wikipedia, The Free Encyclopedia, webpage: <http://www.wikipedia.org>.
- [81]. [Wu04] K. Wu, P. Chuang, D.J.Lilja, "An active data-aware cache consistency protocol for highly-scalable data-shipping DBMS architectures", *Proc. of the first conference on computing frontiers on Computing frontiers*, pp:222-234, 2004.
- [82]. [Yang03] B. Yang and H. Garcia-Molina, "Designing a super-peer network", In *19th International Conference on Data Engineering*, pp: 49-60, 2003.
- [83]. [Zhau04] W.Zhau, C. Meinel, "Implementing role based access control with attribute certificates", *Proc. Of the 6th International Conference on Advanced Communication Technology(ICACT 2004)*, vol. 1, pp:536-541, 2004.
- [84]. [Ziad02] T.Ziadi, B.Traverson, J.M. Jezequel, "From a UML Platform-Independent Component Model to Platform Specific Component Models", *WiSME@UML'2002, Workshop in Software Model Engineering*, webpage: <http://www.metamodel.com/wisme-2002/papers/ziadi.pdf>, 2002.

APPENDIX

Interoperability and porting issue with MVM toolkit

We adopt Model Driven Architecture (MDA) for our design approach for MVM toolkit v.0.1.0. Our components in *components VM* are architecture-independent abstract models, and they can be sent, received and scheduled on heterogeneous environments in a transparent way by using web services. Our components specify the virtual topology, parallel communication patterns, and resource characteristics in a platform independent way. Thus, our components can be thought as Platform Independent Models (PIMs) of MDA. The *components VM* maps these PIMs into Platform Specific Models (PSMs). The implementations of *components VM* differ with machine architectures. However, the implementations on heterogeneous environments should provide the same semantic view and interface to the upper layer, and hide the complexities of the underlying systems. The MVM toolkit v.0.1.0 maps PIMs into Linux-based models. We aim to extend our implementation for Solaris, Sun OS and WIN32 system environments. The main issues of the porting of MVM toolkit v.0.1.0 to different architectures are as follows:

1. I/O operations and system calls
2. Thread and IPC
3. Data type and endian type
4. Compiler and linker
5. Network protocol, socket, etc.

Generally, porting Linux-based programs to Solaris is a simple task since both Solaris and Linux are based on UNIX. We have not modified the Linux kernel for performance optimization in the current version of our toolkit, thus major modification is not required to port our Linux-based implementation into Solaris-based implementation. Although the syntax of some system calls and I/O operations are slightly different from each UNIX-based system, we can avoid this problem by including the compiler options for different

syntaxes on different architectures. The MVM toolkit distributes the source-code level jobs and invokes the local compiler to compile the jobs for the system specific environment. For instance, the local UNIX-compliant compiler can specify an endian type and apply the different rules for different endians. Following is a code snippet in /usr/include/endian.h of Fedora Core 3 Linux, which applies the different rules for different endians.

```
#if __BYTE_ORDER == __LITTLE_ENDIAN
# define __LONG_LONG_PAIR(HI, LO) LO, HI
#elif __BYTE_ORDER == __BIG_ENDIAN
# define __LONG_LONG_PAIR(HI, LO) HI, LO
#endif
```

The compiler types and the size of data types can be specified in a similar manner. However, we still have to solve the data type issue when we pack the data and send them to different architectures. For instance, the size for the long type of Linux IA-32 is four while the size for the long type of Linux IA-64 is eight. We could avoid this problem by the data type packing in a textual form. According to our benchmark, it gives an overhead for message passing. This issue will be handled in the future version of our toolkit. Our implementation of thread and IPC follows the POSIX standard. Thus, the porting of thread and IPC modules can be achieved in the most POSIX compliant systems.

We have not yet implemented our toolkit for WIN32 systems. The requirements of the implementation for WIN32 systems are as follows:

1. TCP/IP and our protocol set should be supported. The BSD socket library for our implementation should be ported to WIN32 socket library. We have implemented the socket communication modules by inheriting our basic socket library. Thus, we plan to extend our basic socket library for WIN32 systems, allowing the most implementation of the communication modules to be reused.
2. System calls and low-level I/O operations should be rewritten for WIN32 systems and provide a same semantic of the Linux-based MVM toolkit.

3. Third party libraries, such as gSOAP and POSIX IPC packages, and GNU compiler/linker should be supported. The Cygwin⁷ toolkit is recommended to compile and link for the WIN32 implementation of the MVM toolkit, as some of our script files are not successfully interpreted on WIN32 shells.

⁷ Cygwin toolkit provides a Linux-like environment for Windows. It is available at <http://www.cygwin.com>

VITA AUCTORIS

NAME	Dohan Kim
COUNTRY OF BIRTH	South Korea
YEAR OF BIRTH	1972
EDUCATION	Bachelor of Science in Physics Kyung Hee University Suwon, Kyung Gi, South Korea, 2000 Master of Science in Computer Science University of Windsor Windsor, ON, Canada, 2005