Middleware Systems for Enabling Users to Adapt to Dynamic Changes in Execution Environments

実行環境の動的な変化にユーザが適応することを可能にするための ミドルウェアシステム

Kenji Kaneda

**Submitted to Department of Computer Science,
Graduate School of Information Science and Technology,
The University of Tokyo on December 16, 2005
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy**

**Thesis Supervisor: Akinori Yonezawa 米澤 明憲
Professor**

Abstract

We designed and implemented three middleware systems for efficient use of dynamically joining and leaving computing resources.

Clusters of computers and computational grids, which have become increasingly important, are dynamic environments. Because of factors such as resource sharing and failure, the quality of resources available to an application changes constantly. For example, since non-dedicated machines such as desktop PCs are shared by multiple users simultaneously, availability of their resources varies dynamically. For systems gathering hundreds of nodes, machine or network failures occur frequently. These dynamic changes in the availability of resources are a great problem for the deployment of clusters and computational grids.

Our goal is to allow users to adapt to changes in their execution environments and use dynamic resources. Towards this goal, we took two approaches with different foci. One approach focuses on building a platform for parallel programming on large-scale geographically-distributed resources. The other approach focuses on providing a single system image for simplifying the utilization of dynamic resources. For the former approach, we designed and implemented a Grid-enabled message-passing library and a command shell for remote job submission. For the latter approach, we developed a virtual machine monitor for providing a single system image.

The Grid-enabled message-passing library is based on a programming model called *Phoenix* which provides a collection of logical machine identifiers that a programmer dynamically allocates (or de-allocates) to (or from) physical machines. With this dynamic allocation of logical identifiers, the library supports nodes dynamically joining and leaving computation during runtime. In addition, the system supports message routing between nodes not directly reachable due to firewalls and/or network address translation (NAT). It also supports resource discovery, facilitating ease of configuration that allows nodes without static names (e.g., DHCP clients) to participate in computation without additional work. To implement these mechanisms, our system runs a distributed resource discovery and routing table construction algorithm, rather than assuming all such pieces of information are available in a static configuration file or similar form. To improve the performance of the routing algorithm, we devised a technique for eliminating

the redundant transmission of routing update messages. We measured the performance of the routing algorithm using 400 nodes in three LANs. The experimental results show that the elapsed time of routing table construction by our algorithm is only about twice as long as that of off-line route calculation.

The command shell enables users to access hundreds of remote computers spread over wide area networks (WANs). In the shell, users can submit jobs to their remote machines, with functions such as input/output redirection and network pipes, over the remote machines. In addition, the shell allows users to easily access their machines behind firewalls and NATs if routes to the target machines exist. We implemented this system with our message passing library. We ran the system on approximately 100 nodes (270 CPUs) and showed its high usability.

The virtual machine monitor virtualizes a shared-memory multi-processor machine on a network of computers. This functionality greatly simplifies use of distributed environments. For example, it enables legacy applications (e.g., Linux kernel for symmetric multiple processor) installed on multi-processor systems to run on a number of less expensive machines. The system supports dynamic addition or removal of machines by allocating one or more virtual processors to a physical processor and by changing the allocation dynamically. We conducted experiments in which parallel coarse-grained tasks were executed inside a virtual eight-way multi-processor machine built on top of eight physical machines. The experimental result demonstrated the feasibility of our approach.

論文要旨

本論文は，動的に参加・脱退する計算資源を効率的に利用するためのミドルウェアシステムについて述べる．

近年その重要度が増しつつある，計算機のクラスタやグリッドは，動的に変化する環境である．資源共有や故障などの原因によって，アプリケーションが利用可能な資源の質は絶えず変化する．例えば，デスクトップ PC などの非占有マシンは複数の利用者によって共有されるため，その可用性が動的に変化する．数百のノードからなるシステムにおいては，マシン・ネットワークの故障が頻繁に起こり得る．このような資源の可用性が動的に変化することが，クラスタやグリッドの普及への大きな障害となっている．

本研究の目標は，利用者が自分の実行環境の変化に適応し，動的に変化する資源を有効利用することを可能にすることである．この目標に向けて，我々は，それぞれ異なる焦点を持つ 2 つのアプローチで取り組む．一つのアプローチは，地理的に分散した大規模環境上での並列プログラミングのための枠組みを構築することに焦点を置いている．もう一つのアプローチは，単一システムイメージを提供し，動的に変化する資源を簡便に利用可能にすることに焦点を置いている．前者のアプローチのために，我々は，Grid-enabled メッセージパッシングライブラリと，遠隔ジョブ投入のためのコマンドシェルを設計・実装した．後者のアプローチのために，単一システムイメージを提供するための仮想マシンモニタを開発した．

Grid-enabled メッセージパッシングライブラリは，*Phoenix* プログラミングモデルに基づく．このモデルは，プログラマが動的に物理マシンに割り当て・解放できる論理マシン識別子の集合を提供する．この論理識別子の動的割り当てによって，ライブラリは，実行時におけるノードの動的な参加・脱退を扱うことができる．さらに，このシステムは，ファイアウォールや NAT などの制限によって直接通信出来ないノード間でのメッセージ配送を扱うことができる．資源発見機構によって，静的な名前を持たないノード（例えば DHCP クライアント）が，手間を必要とせずに計算に参加することも可能になっている．これらの機構を実装するために，我々のシステムは，利用可能な情報が静的に設定ファイルなどの形で与えられると仮定はせずに，分散資源発見とルーティング表の構築アルゴリズムを実行する．ルーティングアルゴリズムの性能向上のために，我々はルーティング更新メッセージの重複転送を除去する技術を考案し，そのアルゴリズムの性能評価を 3 つの LAN にまたがる 400 ノードを用いて行った．その結果，我々のアルゴリズムにおいてルーティング表の構築にかかる時間が，経路情報が静的に与えられた場合の時間の約 2 倍に収まることが示された．

コマンドシェルは，利用者が WAN 上にちらばった数百の遠隔マシンにアクセスするこ

とを可能にする．このシェルでは，遠隔マシンをまたがる入出力のリダイレクションやネットワークパイプなどの機能を用いながら，利用者は遠隔マシンへのジョブ投入を行うことができる．さらに，このシェルでは，何らかの経路が存在しさえすれば，利用者はファイアウォールや NAT の内側のマシンにもアクセスできる．このシステムを，我々のメッセージパッシングライブラリを用いて実装した．このシステムを約 100 ノード（270CPU）上で動作させ，高い有用性を示した．

仮想マシンモニタは，ネットワークでつながれた複数のマシン上に，共有メモリ型マルチプロセッサマシンを仮想的に構築する．この機能によって，分散環境を簡便に利用することが可能となる．例えば，マルチプロセッサマシン上にインストールされる既存のアプリケーション（例えば SMP 用の Linux カーネル）を，コストのより低い複数のマシン上で動作させることが可能になる．このシステムは，一つ以上の仮想プロセッサを一つの実プロセッサに割り当て，さらに，その割り当てを動的に変更させることによって，動的なマシンの追加・削除を扱うことができる．我々は，8 台の物理マシン上に仮想的に 8-way のマルチプロセッサマシンを構築し，その上で粗粒度タスクを並列に実行した．この実験の結果は，我々のアプローチが現実的に可能であることを示している．

Acknowledgments

I would like to thank many people who helped my study and life in general.

First of all, my deepest appreciation goes to my shepherd Professor Akinori Yonezawa. I would like to thank him for his invaluable supports and directing me to this interesting research area. I am indebted to him for the opportunity of working in a first-class research environment.

Associate Professor Kenjiro Taura has also been a great mentor since I was a student, both undergraduate and graduate. He suggested the direction of my work and supported me with his bright ideas — *Phoenix* in particular — and a broad range of background. He significantly contributed towards improving this work. This work would not be possible without his help.

Mentors of the educational program "Professional Programme for Strategic Software", which I attended for two years offered various opportunities to learn and think about relevant topics in different research areas. In particular, I would like to thank Professor Kei Hiraki and Associate Professor Mary Inaba for stimulating me as well in positive and important ways.

I also would like to thank my thesis committee, who made this thesis possible. They gave me various useful advice to improve the quality of the thesis.

I cannot forget to thank colleagues in Yonezawa (TAPLAS) Group and Taura Group for their daily research discussions and assistance. Dr. Yutaka Oiwa and Tosiyuki Maeda supported me with their deep knowlege about Linux and IA-32. Members of the Phoenix Grid Computing and Tools project — Dr. Toshio Endo, Yoshikazu Kamoshida, Yuuki Horita, and Hideo Saito in particular — spent significant energy to improve various aspects of the Phoenix library. Members of the Virtual Infrastructure for Networked Computers (VINCS) project — Dr. Yoshihiro Oyama and Koichi Onoue in particular — motivated me to continue with the research during the difficult times when progress was slow. I dearly thank Ms. Keiko Okada for kindly supporting my day-to-day requests without complaining.

Communications with Professor Joshua LeVasseur, Professor Kazuhiko Kato, Professor Satoshi Matsuoka, and Dr. Kazuyuki Shudo helped me understand relationships of my work to theirs. Anonymous referees for the Journal of Future Generation Computer

Systems, ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, IEEE/ACM International Symposium on Cluster Computing and the Grid, IEEE International Parallel and Distributed Processing Symposium, IPSJ Computer System Symposium, and IPSJ Transactions on Advanced Computing Systems gave insight comments to all the technical details.

Financial support for this thesis was provided by "Precursory Research for Embryonic Science and Technology" of Japan Science and Technology Agency, "Professional Programme for Strategic Software", and Japan Society for the Promotion of the Science (JSPS).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Through the advent of high-speed network technologies, clusters of computers and computational grids — infrastructures for a large number of geographically distributed resources — are becoming indispensable to computational approaches for problem solving [Buy99, FK99]. These clusters and grids enable large-scale coordinated use and sharing of resources, often with a high-performance orientation. For example, aggregating hundreds of computers benefits computing-intensive applications (e.g., simulation system for atmospheric prediction [TSTS03]) as well as data-intensive applications (e.g., web search engines [BDH03], analysis of patabyte-scale archival data of astronomical observatories [YTS04]).

Despite the great promise of such clusters and computational grids, generalizing the use of these systems requires overcoming many issues. One of problems that hinders the wide deployment of clusters and grids is *dynamic change in the quality of resources*; the quality of resources available to individual applications changes constantly because of factors such as resource sharing and failure. For example, machine or network failures are frequent event for clusters and grids gathering hundreds of nodes [1]. Multiple users may be competing for non-dedicated resources while parallel applications are executing. As a result, applications — long running ones in particular — heavily suffer from dynamic changes of resource availability since they frequently encounter such changes in the middle of computation, and restarting computation can be expensive.

To address this problem, applications designed to execute on clusters and/or computational grids should support *adaptive control of parallelism* and *fault tolerance*. Support of adaptive parallelism allows applications to respond easily to addition and/or removal of machines [GK92]. For example, upon receiving notification of addition of an idle

[1] The Google Cluster using about 8000 nodes experienced a node failure rate of 2-3% per year [ABB01].

machine, a parallel application would balance its workloads dynamically by delegating some amount of the workloads to the newly added machine. Fault tolerance guarantees that applications continue operation in the event of unexpected hardware or software failures [Tel01].

We are specifically concerned with support of adaptive parallelism. This is because we believe that supporting adaptive parallelism is feasible in practice and is adequate as a building block towards fault tolerance. Support of adaptive parallelism and fault tolerance are complementary; abilities required for adaptive control of parallelism cannot be provided solely by support of fault tolerance. For example, fault tolerance does not provide a way for handing machine addition. Furthermore, machine removal should be handled in a more efficient manner than traditional roll-back protocols (e.g., checkpointing, message logging) [EAWJ02]. While a basic recovery process need stop the execution of all running processes, a method that allows processes to leave computation without stopping their entire execution process would be preferred.

1.2 Goal of Thesis

Our goal is to design and implement middleware systems that support adaptive parallelism. The middleware systems accommodate dynamic addition and/or removal of machines by abstracting dynamic changes of resources. The systems should have:

Programmability Interfaces the systems provide should not be designed for specific applications but should be flexible and general enough to support a wide variety of real-world parallel applications, including parameter-sweep applications, parallel divide-and-conquer applications (e.g., tree searching), and array-based applications (e.g., LU factorization).

Scalability The middleware systems should be implemented in a scalable fashion. Thus, several issues for guaranteeing scalability need to be addressed. For example, the systems must work without requiring bottleneck modules such as a central server that manages all participating nodes. For another example, the systems should support addition and/or removal of nodes without stopping the entire computation. Moreover, they should allow multiple nodes to be added to and/or removed from computation simultaneously.

Usability The middleware systems should allow users to run their programs in parallel as if the programs were only being run on local computers. More specifically, the middleware systems should allow legacy parallel code to adapt to dynamic changes in its execution environments and to continue operation with little or no modification to the code.

1.3 Previous Work on Middleware Systems for Adaptive Parallelism

We reviewed several main areas of previous work on middleware systems for supporting dynamic addition and/or removal of machines, and determined features that they lack for meeting our goal. We classified these middleware systems into two categories: parallel systems that support adaptive parallelism at *the programming language or library level*, and distributed systems that provide remote process execution and/or process migration to support adaptive parallelism at *the process level*.

1.3.1 Parallel Programming Systems

While many parallel programming languages and libraries have been developed so far, they are insufficient for supporting adaptive parallelism; they lack either general programming interfaces or efficient implementation that can scale to a large number of (> 100) nodes.

Message Passing Interface. Message Passing Interface (MPI) [Mes] is a widely used communication library for parallel and distributed computing, especially for high performance computing. MPI-1 [Mes94] provides a simple flat name space. When an application runs with N processes, the system gives the processes unique names, $0, \ldots, N$, to identify them. The processes interact by sending (or receiving) messages to (or from) one another using their names. Adding to this basic model, MPI-2 [Mes03] provides interfaces that allow dynamic creation of processes after an application has started (e.g., `MPI_COMM_SPAWN`) as well as interfaces for establishing connections between newly created processes (e.g., `MPI_COMM_OPEN` and `MPI_COMM_ACCEPT`). These mechanisms were introduced mainly for the purpose of master-slave model parallel applications.

Both of MPI-1 and MPI-2 provide little or no support for situation where nodes leave or join computation at unexpected times. MPI-1 is not suitable for dynamic environments since it assumes the number of physical processors is fixed. Node names are statically bound to user processes and do not allow programmers to change them dynamically. MPI-2 faces difficulties in supporting general-purpose applications in dynamic environments although it is sufficient for applications with simple communication patterns. For example, explicit connection management with `MPI_COMM_CONNECT` and `MPI_COMM_OPEN` is not flexible enough for neither point-to-point communication between any two nodes nor collective operations (e.g., broadcast, all-to-all communication).

Extensions of MPI for Adaptive Load-balancing. Extensions of MPI with support of adaptive load-balancing include Adaptive MPI (or AMPI) [HLK03] and Dynamic Mes-

sage Passing Interface (Dyn-MPI) [WLNL03].

AMPI allows applications to adapt to dynamic changes of resources with less modification to their code. More specifically, physical processors are no longer visible to programmers in AMPI; programmers use a large number V of virtual processors, independent of the number of physical processors P. In contrast to standard MPI programs that divide computation into P processes, one for each of the P physical processors, an AMP programmer divides the computation into V virtual processors that correspond to individual user processes. The runtime of AMPI creates multiple user processes per each physical processor and then, given different physical processor loads, migrates the user processes as appropriate.

Since AMPI provides no sophisticated method to control the granularity of virtual processes, programmers can have a difficult time creating an appropriate number of virtual processors; both inappropriate increases and decreases of the granularity can lead to poor performance. For example, fine-grain approaches may have significantly more messages than their coarse-grain counterparts[2]. In addition, virtualization incurs overhead costs for reasons such as process creation and management. On the other hand, coarse-grain approach may cause load imbalance because of an inflexible allocation of user processes.

Dyn-MPI is another extension of MPI. While the ultimate goal of Dyn-MPI is same as AMPI, the approach that Dyn-MPI takes is different. Dyn-MPI supports adaptive parallelism by automatically re-distributing data on the fly when changes occur in the application or the underlying environments. To support this facility, interfaces of Dyn-MPI have several extensions, including relative ranks that vary over the course of computation when nodes are added to or removed from computation.

To support such automatic data re-distribution, Dyn-MPI restricts the kinds of applications it can support. More specifically, Dyn-MPI is designed only for iterative applications consisting of one or more phases, which are sections of code comprised of computation followed by communication. In addition, programmers basically need to identify data to be considered for redistribution.

Note that much of the work on supporting fault tolerance for MPI (e.g.,FT-MPI [FD00], MPI/FT [BNC+01], MPICH-V [BBC+02]) is not sufficient for our purpose. As mentioned in Section 1.1, these systems have inherent difficulties with dynamic accommodation.

Parallel Virtual Machine. Parallel Virtual Machine (PVM) [GBD+94] is a software system that permits a heterogeneous collection of computers to be viewed as a single parallel computer from a user program. More specifically, PVM users write their applications as

[2]Imagine a nearest neighbor communication pattern in which an individual node needs to send one message per boundary edge. In this case, a fine-grain approach greatly increases the number of exchanged messages.

collections of cooperating tasks. Standard interfaces provided by PVM include initiation and termination of tasks across networks as well as communication and synchronization between tasks.

Although PVM provides several interfaces for dynamic addition and/or removal of tasks and hosts (e.g., `pvm_addhosts` and `pvm_delhosts`), it suffers from the same drawbacks that interfaces of MPI have. First, since an identifier called a task identifier (TID) is statically bound to a newly created task[3], programmers face difficulties in migrating TIDs across different machines according to dynamic changes of available machines. Moreover, PVM requires programmers to deal with these explicitly and to implement their own method for reassigning work to other tasks. Doing so is very cumbersome and may be inefficient for applications involving frequent addition or removal of machines.

Note that although the first problem can be solved by extensions of PMV with support of process migration (e.g., MPVM) [CCK+95], the second problem still remains unsolved.

Libraries Based on Bulk Synchronous Parallel Models. Several systems based on a master-worker style programming model called the Bulk Synchronous Parallel (BSP) model [Val90] have been developed. These systems include Bayanihan [Sar99], a Java-based framework for volunteer computing systems. Bayanihan allows high-performance parallel computing inexpensively by enabling ordinary Internet users to share the processing power of their idle computers.

In BSP, a parallel program runs across a set of virtual processors and executes as a sequence of parallel supersteps separated by barrier synchronizations. Each superstep is composed of three ordered phases: the local computation phase, the global communication phase, and the barrier synchronization phase.

Drawbacks of BSP model are as follows. First, the model is not suited to implement an efficient point-to-point communication facility. For example, the global communication phase can be a serious and unnecessary sequential bottleneck in cases where such restrictions can be removed. Second, the model allows nodes to join and leave computation only when programs reach the barrier synchronization phase.

Divide-and-Conquer Systems. Many programming languages designed for divide-and-conquer parallelization have been developed. These systems include Cilk-NOW [BL97], Satin [vNKB01], Javelin 2.0 [NPRC00], Atlas [BBB96], Dynasty [BPZ96], and DCPAR [FK95]. In these systems, programmers basically annotate potential parallelism in the form of *spawn* and *sync* constructs, and then distribute parallel tasks over machines with scheduling mechanisms such as random work-stealing.

These systems are designed only for kinds of parallel applications that have directed acyclic graph (DAG) dependency between tasks; they are not suitable for general-purpose

[3] According to [GBD+94], the TID of task t contains a host number indicating a location where t runs.

parallel applications.

Peer-to-Peer Content Distribution Technologies. Much work has been carried out to build scalable self-organizing infrastructures for providing various services, such as persistent storage, file sharing, and DNS lookup (e.g., Chord [SMK+01], CAN [RFH+01], Pastry [RD01], Tapestry [ZKJ01], and Kademlia [MM02]). To implement such functionalities in a scalable manner, nodes act autonomously and construct overlay networks without requiring intermediation or support of a centralized authority. In addition, nodes re-construct overlay networks adaptively to tolerate failures both in network connections and computers, as well as a transient population of nodes.

These systems provide a large and fixed name space abstraction, mediating communication to implement services. They all build a routing infrastructure so that involved nodes can send messages to any name. For example, some peer-to-peer file-sharing systems use distributed hash-table (DHT) routing [ATS03] to map file names to their locations and to allow nodes to send requests (e.g., file insertion, deletion) to other nodes where specified files are located.

The peer-to-peer technologies cannot be applied straightforwardly to building middleware systems for a wide variety of parallel applications. For example, most DHT routing algorithms are not suitable for high performance computing since insertion or deletion of items requires $O(\log n)$ hops message delivery where n is the number of nodes.

1.3.2 Distributed Systems with Remote Process Execution and/or Mobile Processes

We reviewed distributed systems that support adaptive-parallelism by providing services such as remote process execution and/or mobile processes.

Task Scheduling Systems. Task scheduling systems aim to take advantage of 2available processing power (CPU cycles) of computers in distributed environments. This is achieved by breaking down a computer-intensive task into small work units, distributing them to multiple computers, and gathering the results. A single server or several central servers balance workloads adaptively by spawning tasks only to idle computers. Practical examples include systems designed for computers inside a single local-area network (e.g., Load Sharing Facility [Pla], Portable Batch System [Opea], Condor [LLM88]) as well as for wide-area environments (e.g., Ninf-G [TNS+03], Condor-G [FTF+01], Nimrod/G [BAG00], XtremWeb [CDF+05], SETI@home [SET], DCGrid [Ent], Grid MP [Uni]).

While these systems are useful for scheduling loosely-coupled or independent tasks, they are not appropriate for general parallel applications for several reasons. First, since

central coordination is invariably required for distributing tasks and collecting the results, servers can become bottlenecks under dense communication among participating nodes. Furthermore, these systems lack sophisticated methods for point-to-point communication between clients. For instance, most of these systems do not provide a mechanism that allows individual clients to identify one another. Thus, the clients need to somehow obtain information about other participating nodes (e.g., IP addresses, DNS hostnames) and assig unique identifiers to all the nodes explicitly to communicate with one another.

Process Migration Technologies. Process migration [MDP+00] is the ability to transfer a process from one machine to another. It is a useful facility in distributed environments, and its potential benefits include adaptive load balancing and fault resilience by migrating processes running on faulty hosts. Operating systems and middleware systems that support process migration include Accent [RR81], MOSIX [BL98], Sprite [OCD+88], V [Che88], and Zap [OSSN02]. Added to process migration, systems that support migration of virtual machines have been developed recently (e.g, Xen [CFH+05], VMotion [NLH05], Quasar [OOY05]).

These systems suffer from several drawbacks. First, these systems face difficulties in load balancing like AMPI. Appropriately controlling the number of processes that maximizes performance is difficult. Second, the implementation of inter-process communication is inefficient. In most of the systems, IPC is typically handled by forwarding requests to a home node on which the process originated. Although some systems (e.g., Zap) support persistent communication over mobile processes without leaving behind any residual state after migration, these systems cannot migrate both end-points of a connection simultaneously [SN02].

1.4 Our Approach and Contributions

Our basic approach to supporting adaptive parallelism is based on a traditional resource-virtualization methodology in computer science — abstracting physical resources into a large, fixed number of virtual resources as viewed by users. Our middleware systems map virtual resources to physical resources in the following manner (See Figure 1.1):

- The systems initially create a larger number of virtual resources than physical resources, and map one or more virtual resources to a single physical resource.
- The systems change the mapping dynamically, adapting to dynamic changes of physical resources.

More specifically, we designed and implemented two middleware systems that virtualize *node names* used for specifing message destinations and *processor units*. The system

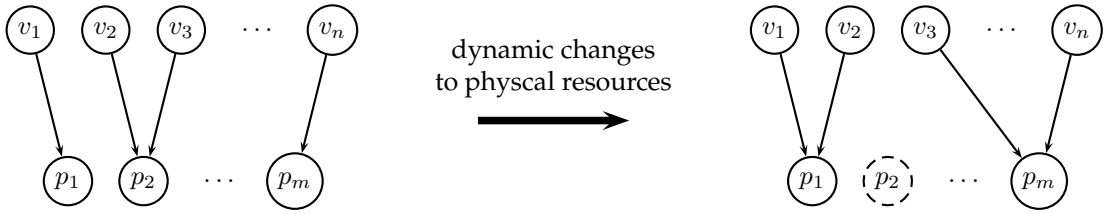

Figure 1.1: Resource-virtualization approach to supporting adaptive parallelism. v_i and p_i denote a virtual and physical resource respectively. The number of virtual resources n is larger than the number of physical resources m. A fixed virtual resources is viewed although p_2 is removed.

Table 1.1: Our middleware systems for supporting adaptive parallelism

		Phoenix	**Virtual Multiprocessor**
virtualized resources		node names	processor units
system overview		message passing library	virtual machine monitor
criteria	**programmability**	strong	strong
	scalability	strong	weak
	usability	fair	strong

virtualizing node names is *Phoenix*, a message passing library, and the system virtualizing processors is *Virtual Multiprocessor*, which is based on classic operating system concept of a virtual machine monitor (VMM) [Gol74] (See Table 1.1).

We describe contributions we made through explanation of an overview of the Phoenix library and Virtual Multiprocessor. Our contributions relevant to the Phoenix library are as follows:

- We implemented the Phoenix library, which supports adaptive parallelism [TKEY03]. It is based on the Phoenix programming model that subsumes regular message passing models (e.g., MPI) and supports general-purpose applications. In contrast to traditional message passing systems like MPI, the model provides a collection of virtual node names that programmers dynamically allocates to or de-allocates from processes. With this dynamic allocation of virtual node names, the library supports nodes joining and leaving computation at any time.

  This approach completely differs from SIMD where there are as many threads of control as the number of virtual node names (e.g., AMPI). Virtual node names in Phoenix are just names given to user processes as a way for specifying message destinations. Since each process only has as many threads of control as explicitly created (usually one), Phoenix enhances scalable, efficient and flexible load balanc-

ing.

The Phoenix library is designed for wide-area networks (WANs), of which configurations (e.g., firewalls, DHCP, network address translation: NAT) often restrict communication among processes. To allow applications to be easily deployed under WANs, the library supports message routing between nodes not directly reachable due to firewalls and/or NATs.

We evaluated the performance of the Phoenix library using several benchmark programs, including a parallel ray-tracing program based on Pov-Ray [Pov] and Integer Sort in NAS Parallel Benchmark suite [NASa]. Experimental results indicated applications with a small task migration cost can quickly take advantage of dynamically joining or leaving resources. The parallel ray-tracing that distributes workloads by a divide-and-conquer algorithm achieved a good speedup with a large number of nodes across multiple LANs (about 78 times speedup using 104 CPUs across three LANs).

- To improve the scalability of Phoenix, we devised several techniques for improving the performance of the routing algorithm [KTY04]. These techniques include elimination of redundant transmission of routing update messages. In addition, we extended Phoenix for support of resource discovery, facilitating ease of configuration that allows nodes without static names (e.g., DHCP clients) to participate in computation without specific efforts.

  We measured the performance of the routing algorithm using 400 nodes in three LANs. The experimental results showed the elapsed time of routing table construction by our algorithm is only about twice as long as that of off-line route calculation.

- To demonstrate the feasibility of Phoenix, we implemented Virtual Private Grid (VPG), a command shell for easily and efficiently accessing hundreds of remote machines spread over WANs.

  VPG allows users to submit jobs to their remote machines with facilities, such as input/output redirection and network pipes, over remote machines. In addition, VPG allows users to easily access their machines behind firewalls and NATs if routes to the target machines exist. Moreover, it tolerates dynamic changes of available machines.

  While the original implementation of VPG constructs a self-stabilizing spanning tree to support dynamic addition and removal of machines [KTY02, KTY03], the implementation with the Phoenix library tolerates such changes with dynamic routing table construction. We ran VPG on about 100 nodes (270 CPUs) and showed its high usability.

Our contributions relevant to Virtual Multiprocessor are as follows:

- We designed and implemented Virtual Multiprocessor [KOY05, KOY06]. It virtualizes a shared-memory multi-processor machine on a commodity cluster to facilitate porting of parallel applications as well as operating systems for shared-memory multi-processor machines with no or little modification to the code.

  We built a virtual eight-way multi-processor machine on eight physical machines, with Linux installed. We ran parallel, coarse-grained tasks on the virtual machine. The experimental results demonstrated the feasibility of our approach.

- We extended Virtual Multiprocessor for support of adaptive parallelism. More specifically, we implemented a facility that provides a fixed number of processors even if physical machines are added and/or removed dynamically. It allows parallel applications to adapt to dynamic changes with no modification to their code.

1.5 Structure of Thesis

The thesis is structured as follows. In Chapter 2, we introduce the basic concepts of Phoenix, a message passing library for accommodating computational grids. We describe its APIs and performance evaluation. In Chapter 3, we explain the communication subsystem of the Phoenix in detail. We describe several optimization techniques devised for routing table construction and resource discovery. In Chapter 4, we present Virtual Private Grid, a command shell for utilizing hundreds of remote computers efficiently. This command shell is implemented with the Phoenix library. In Chapter 5, we explain the design, implementation, and evaluation of Virtual Multiprocessor, a virtual machine monitor for providing a single system image. In Chapter 6, we conclude with general discussions on future work.

Chapter 2

Phoenix: A Grid-Enabled Message Passing Library

Overview

This chapter presents the Phoenix library for accommodating dynamically joining and/or leaving resources. In contrast to existing parallel libraries based on message passing models (e.g., MPI), the Phoenix library has two distinguishing features. First, the library is based on the Phoenix programming model [TKEY03]. The model provides a large and fixed *virtual node name space* of which elements are used for specifying message destination. By mapping virtual node names to processes dynamically, a programmer easily and efficiently distribute workloads over processes that joins and leaves computation. Second, the library supports message routing between nodes not directly reachable due to firewalls and/or NAT. We evaluated the performance of the Phoenix library using several benchmark programs, including a parallel ray-tracing program based on Pov-Ray [Pov] and Integer Sort in NAS Parallel Benchmark suite [NASa]. Experimental results showed the parallel ray-tracing with divide-and-conquer algorithm achieved a good speedup with a large number of nodes across multiple LANs (about 78 times speedup using 104 CPUs across three LANs).

2.1 Introduction

Computational grids are becoming increasingly important infrastructures for high performance parallel computing. Aggregation of a large number of (e.g., > 100) processors enables a wide variety of applications, such as scientific applications, to tackle large-scale problems.

Although a message passing model is a dominant programming model for computational grids, message passing libraries such as MPI suffer from the following drawbacks. First, the message passing libraries lack a suitable programming model for efficiently utilizing *non-dedicated* resources. Since most of resources in grids are shared by multiple users, the quality as well as quantity of the resources available to an application changes constantly. Although it becomes important for parallel programs to adapt to such dynamic changes, writing adaptive parallel programs with traditional message passing libraries requires large efforts. Second problem is that administrative policies often restrict communication between nodes. For example, a machine behind a firewall or NAT cannot accept incoming connections from outside the subnet. If nodes outside the subnet attempts to send messages to the inside, they need to somehow forward messages via bidirectional connections (e.g., TCP connections) established by machines inside the firewall or NAT. As a result, many implementations of message passing libraries that assume direct all-to-all communication do not work correctly across multiple LANs.

To address these two problems, we implemented the Phoenix message passing library. In contrast to existing parallel libraries based on message passing models (e.g., MPI), the Phoenix library has two distinguishing features. First, the library is based on the Phoenix programming model [TKEY03]. The model provides a large and fixed *virtual node name space* of which elements are used for specifying message destination. Basically, an application distributes its data structures over a large number of virtual processors rather than over physical processors (as MPI programs do). By mapping virtual node names to processes dynamically, the application easily and efficiently distribute workloads over processes that joins and leaves computation constantly. Second, the Phoenix library allows applications to be easily deployed under WANs, of which network configurations (e.g., firewalls, DHCP, NAT) often restricts communication among processes. It supports message routing between nodes not directly reachable due to firewalls and/or NAT.

We evaluated its performance using several benchmark programs, including a parallel ray-tracing program based on Pov-Ray [Pov] and Integer Sort in NAS Parallel Benchmark suite [NASa]. Experimental results indicate applications that have a small task migration cost can quickly take advantage of dynamically joining/leaving resources. The parallel ray-tracing that uses a divide-and-conquer algorithm achieved a good speedup with a large number of nodes across multiple LANs (about 78 times speedup using 104 CPUs across three LANs).

The remainder of this chapter is organized as follows. Section 2.2 presents the Phoenix programming model. We describes its basic concepts, APIs and several programming examples. Section 2.3 describes the implementation of the communication mechanism of the library. Section 2.4 presents performance measurements. Section 2.5 discusses related work. The final section summarizes the chapter.

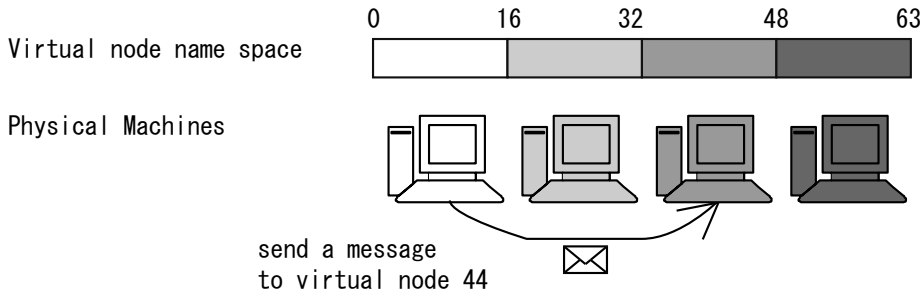

Figure 2.1: An example of message delivery in the Phoenix model

2.2 Phoenix Programming Model

2.2.1 Basic Concepts

Like traditional message passing models, Phoenix provides an application with a flat and per application node name space, which is a range of integers, say $[0, \ldots, L)$. A node name specifies a message destination. Unlike regular message passing, L can be much larger than the number of participating processes and must be *constant* regardless of it. Given L, we call the space $[0, \ldots, L)$ *the virtual node name space* of the application. Since the number of participating processes may not match L, each process assumes, or is responsible for, *a set of* virtual node names.

This virtual node name space has two characteristics to efficiently support parallel applications that change the number of participating processes at runtime. First, given a message destined for a virtual node, the runtime system routes the message to some process that currently assumes the specified virtual node. Figure 2.1 shows an example of message delivery. When a process sends a message to virtual node 44, the message is delivered to another process to which virtual node is mapped. Second, Phoenix allows the mapping between processes and virtual nodes to change at runtime. Since the entire virtual node space nevertheless stay constant, Phoenix supports parallel applications that change the number of participating processes at runtime, while providing a programmer with a simpler view of a fixed node name space.

We comment on how L can be chosen. L can be chosen for an application's convenience, as long as all participating processes can agree on the same value. As we explain in Section 2.2.3, the primary purpose of virtual node names is to associate each piece of application data with a virtual node name, so that mapping will derive data distribution. So a reasonable choice is often determined by the size of the application data to be distributed over processes. For example, if the only distributed data structure used by the application is a hash table with N (constant) keys, we may have $L = N$ and associate hash items of key x with virtual node x. For another example, Figure 2.2 illustrates how

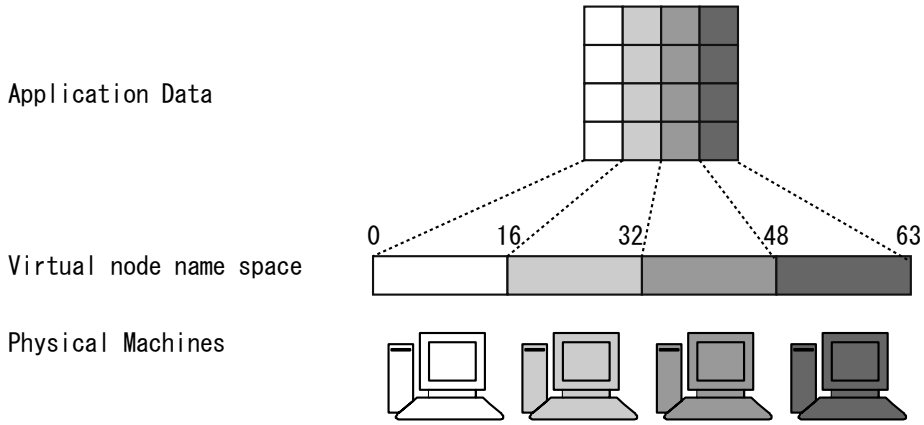

Figure 2.2: An example of matrix operation written in Phoenix

a program that operates a matrix is written in Phoenix. The programmer writes the matrix operation by letting L be the number of the elements of the matrix by associating the elements with virtual node names. If there are many distributed data structures of different sizes or even unknown sizes, one can simply choose an integer much larger than any conceivable number of data items, say 2^{62}.

As will be clear from the above description, processes participating in a Phoenix application should cooperatively cover the entire virtual node space. More specifically, Phoenix applications should maintain the following conditions:

- No two processes assume the same virtual node at any instant.
- There may be an instant at which no processes assumes a virtual node, but in such cases, one must eventually appear that assumes it.

The intent is to always maintain the invariant that the entire virtual node name space is disjointly covered by participating processes. We, however, slightly relax this condition (the second bullet), allowing finite periods of time in which no process assumes a virtual node. Messages to such a virtual node are queued until one appears that assumes it, rather than lost, bounced, or redirected to a process in a way the programmer cannot predict. This is important for supporting applications that migrate application-level states from one process to another, and/or applications that allow processes to permanently leave.

2.2.2 Programming Interfaces

We describe simplified version of programming interfaces provided by the Phoenix library. These APIs are summarized in Table 2.1. The actual APIs for the C language,

Table 2.1: Basic Phoenix APIs

Message send/receive functions:

`ph_send(`v, m`)`	sends message m to virtual node v
m `= ph_recv()`	receives a message destined for assumed virtual nodes and returns it

Virtual node name mapping functions:

`ph_assume_vps(`V`)`	assumes a set of virtual nodes V
`ph_release_vps(`V`)`	releases a set of virtual nodes V

Initialize/finalize functions:

`ph_initialize(`p, f`)`	initializes the Phoenix runtime with loading messages from log file f and listening to port p
`ph_add_port(`p`)`	adds p to possible contact points
`ph_finalize(`f`)`	finalizes the Phoenix runtime with logging messages at file f

Miscellaneous functions:

v `= ph_get_resource_name()`	returns a unique resource name

which is slightly more verbose, are shown in Appendix A.

Message Send/Receive Functions. `ph_send()` sends a message to a specified virtual node. If a destination virtual node v is mapped to some process p, the message is delivered to p. If v is not currently mapped to any processes, the message is enqueued to the sender's local message queue. It remains in the queue until v is mapped to some process.

`ph_recv()` looks up caller process' local message queue and dequeues a message destined for any element in a set of virtual nodes mapped to the caller.

Virtual Node Name Mapping Functions. `ph_assume_vps()` and `ph_release_vps()` change the mapping between processes and virtual nodes. When a process p assumes new virtual nodes V, the system begins to deliver messages destined for some v in V to p. When a node releases virtual nodes, the system stops delivering messages destined for the virtual nodes.

Initialize/Finalize Functions. `ph_add_port()` informs the Phoenix runtime of ports at which the node accesses other nodes. By using this information, each node establishes connections to construct an overlay network. A user lists such accessible ports according to security polices as follows. If a node has only a private IP address, connections to this node from outside its subnet will not be listed. Similarly, connections *to* DHCP clients will not be listed, but those *from* them will. Note that this information needs not be very precise. It does not affect correctness to add non-existing ports. Similarly, it does not affect correctness to add connections that are actually blocked, or not to add connections that are actually possible. In current implementation, Phoenix establishes connections with either TCP, Secure Shell tunneling [Opeb], or Secure Socket Layer [Oped].

`ph_finalize()` shuts down the system for temporal or permanent leave of nodes. A node tries to leave after removing messages destined for other virtual nodes from its local queue and sending them to its neighbors. It also tries not to receive further messages by pretending that that no virtual nodes are reachable via the node. Note that Phoenix does not ensure that the local queues always eventually become empty. For example, if every node begins to leave simultaneously, it cannot necessarily make its queues empty. Thus, in current implementation, Phoenix allows nodes to store remaining messages at a local disk if they want.

`ph_initialize()` initializes the system. When this function is called, the system begins establish connections to construct an overlay network. It also begins to load messages stored by `ph_finalize()` if necessary.

Miscellaneous Functions. `ph_get_resource_name()` gets a resource name outside a virtual node name space (e.g., resource names are in $[2^{63}, 2^{64})$). Resource names are bound to individual processes and do not change during program execution. The name is used not for normal application messages, but for special protocols such as application state migration protocol [TKEY03]. An Example of programs that use resource names for migration is shown in Appendix B.

Collective Operations. Although the library provides no primitive for collective operations (e.g., broadcast, reduction), some of the collective collections can be emulated with the combination of the unicast operations such as `ph_send` and `ph_recv`.

Figure 2.3 illustrates pseudo-code of an example implementation of multicast operation. Given a range of virtual nodes $I = [l, u)$, a process assuming virtual node $root$ initiates multicast and delivers $content$ to all processes assuming virtual node $v \in I$. To deliver $content$ to the nodes at least once, information about virtual nodes that have not yet received $content$ is attached to each message. When receiving a message, a process forwards it to virtual nodes that have not yet received $content$ according to the attached information. Specifically, if the message indicates that virtual nodes I' have not yet re-

```
Procedure multicast(I = [l, u), root, content):
begin
  (* initiate the multicast *)
  if root ∈ V then begin
    send ⟨[l, u), content⟩ to l;
  end;

  (* wait until messages for all virtual nodes in I ∩ V arrive *)
  R := ∅;
  while R ∩ V ⊂ I ∩ V then begin
    receive ⟨I', content⟩;
    R := R ∪ I';
    (* forwarded the message *)
    multicast_sub(I' \ (V ∪ R), content);
  end;
end

Procedure multicast_sub(I = [l, u), content):
begin
  (* divide I to several (two in this figure) ranges *)
  send ⟨[l, (l + u)/2), content⟩ to virtual node l;
  send ⟨[(l + u)/2), u), content⟩ to virtual node (l + u)/2;
end
```

Figure 2.3: Pseudo code of multicast. A process that assumes virtual node $root$ initiates multicast, and $content$ is delivered to all processes that assume virtual node $v \in I$ at least once. p, V, and R respectively denote a caller process, virtual nodes assumed by v, and a set of virtual nodes attached with messages v has already received.

ceived $content$, the process forwards it to virtual nodes $I' \setminus (V \cup R)$ where V and R are respectively a set of virtual nodes that the process assumes and a set of virtual nodes that the process has already sent $content$ to. We also note that processes forward messages in parallel to reduce overall latency of multicast.

2.2.3 Writing Parallel Programs in Phoenix

The Phoenix model facilitates programming of parallel applications that support adaptive-load balancing for high performance computing (e.g., LU factorization [ETKY04], distributed game-tree search [Kan04]) as well as distributed systems (e.g., a distrusted file

system, massively multi-player online game system [YKaAY05]). This section describes how programs are written in Phoenix, showing two examples: divide-and-conquer algorithm and an integer sort program.

Divide-and-Conquer Algorithm. We shows how divide-and-conquer algorithm with by random work-steal scheduling [BL97] is described in Phoenix. In the random work-steal scheduling, each process has a task queue and executes tasks in its queue. When its queue becomes empty, it tries to steal tasks from other nodes by sending a task-steal request to randomly chosen node. Upon receiving the request, a process gives some tasks in its queue to the sender.

This work-steal scheduling is implemented easily due to the disjoint-cover property described in Section 2.2.1. Since the property implies that a message to any virtual node is eventually delivered to some process, the application simply needs to send a task-steal request to a randomly chosen virtual node. Even if the program does not know processes that are currently participating in computation, requests are eventually delivered to processes that assume destination virtual nodes.

Integer Sort. We show porting of Integer Sort in NAS Parallel Benchmark suite [NASa] (written in MPI) to Phoenix. First, we briefly sketch an overview of the original program written in MPI. It uses a bucket sort algorithm. Given an array of N elements, each elements of which is in a range $[0, M)$, it divides the range into L sub-ranges $R_0 = [0, M/L), R_1 = [M/L, 2M/L), \ldots, R_{L-1} = [(L-1)M/L, M)$. The set of elements in a sub-range R_i is called *a bucket* i and denoted as B_i. The array is block-partitioned, and in regular message passing code with P processors, process 0 sorts the buckets that roughly cover the smallest N/P elements, processor 1 the buckets for next N/P elements, and so on. Then each process collects elements that it should sort as follows:

1. First, each processor counts the number of elements in each its local bucket.
2. Next, each processor broadcasts the counts to all other processors and receives counts from them by `MPI_Allreduce()`. Now all processors know how many elements are in each bucket of the entire array; it can determine which buckets each processor should sort according to the above mentioned assignment.
3. Then, it actually distributes elements in its local buckets to appropriate processors by calling `MPI_Alltoall()` and `MPI_Alltoallv()`.

To re-write the above MPI program in Phoenix, we achieve block partitioning by fixing the size of the virtual node space to the number of buckets (i.e., L). Then we make a process that assumes virtual node i sorts elements in B_i. `MPI_Allreduce()`, `MPI_Alltoall()`, and `MPI_Alltoallv()` are essentially broadcast. In Phoenix, each

process could accomplish broadcast by simply sending a message to each element in L. Since it requires each node to send L messages in total, it obviously causes large overhead if the number of processors is much smaller than L. Thus we accomplish the broadcast as follows:

1. The sender attaches the range $I = [0, L)$ with the message and sends it to virtual node 0.
2. When receiving a message attached with range $I = [p, q)$, a node removes from I all elements that are corresponding to the assumed virtual nodes. The remaining elements are divided into several ranges, and each range is forwarded to the least element in the range (i.e., a message with range $[p, q)$ is forwarded to virtual node p).
3. Each process repeats **step 2** until it receives the entire range that it assumes.

The above method can reduce the number of messages if receiving node assumes many virtual nodes.

2.3 Implementation

This section describes an overview of the implementation of the Phoenix. In particular, we briefly sketch the communication mechanism of the Phoenix — how the runtime system delivers messages to destination virtual nodes. The details of the communication mechanism are described in Chapter 3.

The implementation of the communication mechanism has two features. First, the implementation is completely decentralized and self-organizing. Second, the system allows processes to communicate with one another even if direct point-to-point communication between some nodes is restricted for administrative and/or security policies.

To achieve the features, the system builds an overlay network among participating processes and routes messages via the network. More specifically, the Phoenix runtime acts as follows:

1. The runtime system tries to connect to ports specified by `ph_add_port`. Connections that cannot be established are retried in a fixed interval. The topology of the overlay network is not necessarily a perfect graph.
2. Along with maintaining connections, the runtime systems cooperatively construct a routing table by exchanging routing information. Each process announces virtual nodes it assumes, and announcements are propagated to other nodes.

   Among many proposed routing table construction protocols, we currently employ Destination-Sequenced Distance Vector routing (DSDV) algorithm [PB94] originally

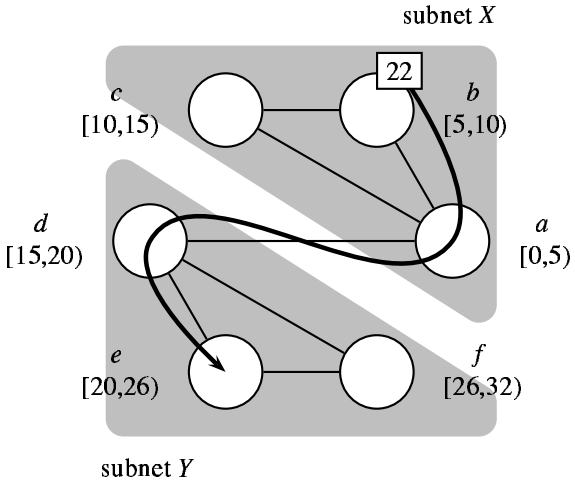

Figure 2.4: Process of message delivery: node b sends a message to virtual node 22.

proposed in the context of mobile ad-hoc networks. It was chosen because it consumes a relatively small amount of memory compared to other schemes based on distance-vector and is relatively simple to implement.

3. When the `ph_send` function is called, the runtime forwards a message to its preferred neighbor by looking up its routing table

4. Phoenix automatically re-constructs an overlay network whenever the topology of the underlying network changes. Whenever the nodes join/leave or physical links become broken, machines add/remove connections to keep the network connected.

Figure 2.4 illustrates how the Phoenix runtime forward messages. Nodes in Figure 2.4 indicate physical machines. Each node's label indicates its hostname and virtual nodes assumed by a process running on the machine. There are two subnets X and Y. Node a, b, and c belong to subnet X while d, e, and f belong to subnet Y. A firewall blocks restrict communication across X and Y except communication between a and d. Because of this restriction imposed by the firewall, an overlay network is built as indicated by the solid lines in Figure 2.4. Suppose b sends a message to virtual node 22, which is assumed by e. Since there is no direct connection between b and e, the message is forwarded via b, a, d and e as indicated by the arrow in Figure 2.4.

Table 2.2: Experimental environments

	Nodes	CPU	Number of CPUs	Interconnect
Cluster A	SunBlade 1000 Cluster	750MHz	2 CPU $\times$ 16 nodes	100 Mbps Ethernet
SMP B	SunFire15K SMP	900MHz	72 CPU	shared memory
Cluster C	SunBlade 1000 Cluster	750MHz	2 CPU $\times$ 128 nodes	100 Mbps Ethernet

2.4 Experiments

We studied performance of two applications: a parallel ray-tracing program based on Pov-Ray [Pov] and Integer Sort in NAS Parallel Benchmark suite [NASa]. The ray-tracing uses divide-and-conquer algorithm with random work-stealing scheduling described in Section 2.2.3. Integer Sort is also implemented as mentioned in Section 2.2.3.

2.4.1 Performance on Static Configurations

We measured a speedup of the parallel Pov-Ray both in a single cluster (Cluster C) and across three SMP/clusters in Table 2.2. Nodes within a cluster are connected via 100Mbps switches. Only SSH connections are allowed across LANs. The raw TCP bandwidth (measured by the bandwidth of a large http GET request) between two LANs is approximately 100Mbps, but the actual throughput over SSH is 30-60Mbps. This is clearly a bottleneck for many parallel programs that would scale well within a LAN. For the multi-LANs experiments, we mix CPUs from the three systems in a constant ratio ($1 : 4 : 8$). Pov-Ray draws a picture of 8000 by 250 pixels, taking approximately two hours on a single CPU.

Figure 2.5 shows a speedup of the ray-tracing program. Pov-Ray achieved a good speedup both on the single LAN and on the multiple LANs. The experimental results show that random work-stealing scheduling is in fact a very communication-efficient load balancing scheme.

We also measured performance of Integer Sort (problem class C) on a heterogeneous Pentium III cluster consisting of 16 processors of 800 MHz and 16 processors of 1.4 GHz. All nodes in the both clusters are connected with 100 Mbps Ethernet. We allocated virtual nodes equally on each process (the number of elements each processor sorts is roughly equal). Figure 2.6 shows a speedup up to 32 processors and comparison of our Phoenix implementation to its original MPICH [MPI] version. The results indicate that the overhead of the above broadcast mechanism was not so large compared to MPICH, and that the broadcast mechanism can be applied to other parallel programs (e.g., LU factorization that uses block-cyclic partitioning technique [TKEY03]).

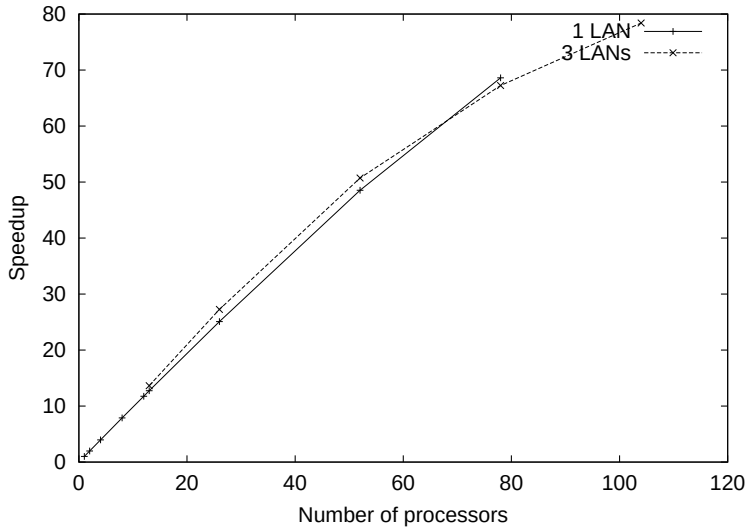

Figure 2.5: Speedup with fixed processors

2.4.2 Performance on Dynamic Configurations

We measured performance of the ray-tracing on Cluster C in Table 2.2 with dynamic configurations. More specifically, we begin with a small number of nodes and add one node at a time in a regular interval, up to 64 nodes. After running with the 64 nodes for a while, then we remove one node at a time. A newly added node sends a join request to a randomly chosen virtual node name. Whichever process received the request splits its range of assuming virtual nodes into two equal ranges and gives the latter half to the requester.

Figure 2.7 demonstrates Phoenix's capability of dynamically adding/removing nodes. We defined a unit progress as a line of the picture whose image has been calculated, and measured the number of unit progresses made in each second. The graph shows "speedup" of each second, which is the number of unit progresses made in the second, divided by the number of progresses per second in a single node run. Also shown as "fixed" is the speedup obtained in the fixed-resource, single LAN experiment for the number of processors participating at that moment. The graph shows Pov-Ray takes advantage of dynamically added nodes very quickly. This is not surprising because they use dynamic load balancing and have relatively small application-level states that need migrate or be copied to new nodes.

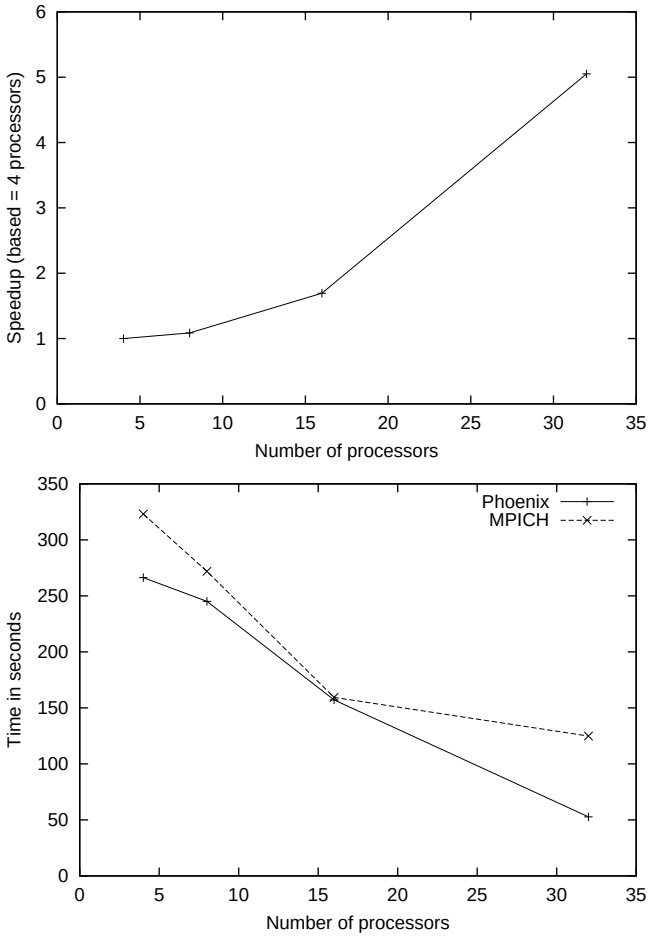

Figure 2.6: Speedup and comparison to MPICH of Integer Sort (problem class C)

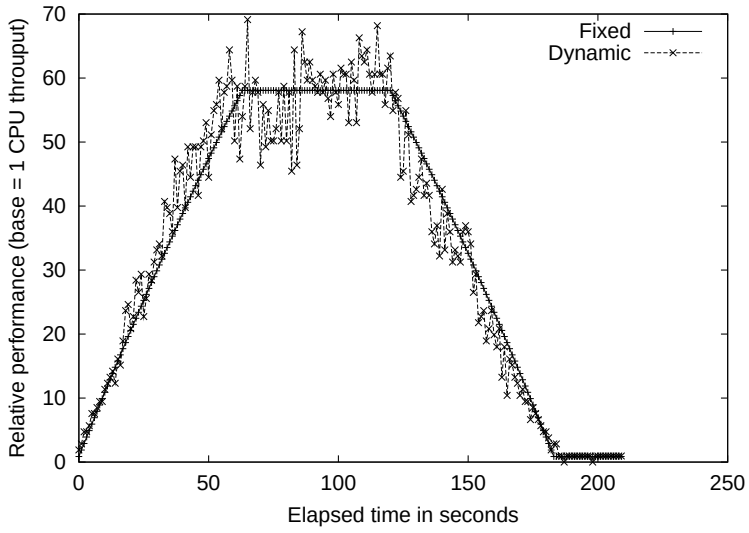

Figure 2.7: Dynamic performance improvement as processors join and leave

2.5 Related Work

Message Passing Systems for WANs. Although many message passing systems for WANs have been developed so far, to our knowledge, none of existing implementation of message passing systems are feasible for several reasons.

Firstly, since their user interfaces and mechanisms for naming are based on traditional message passing model, they face difficulties in handling machine addition and/or removal. For example, although extensions of MPI for fault tolerance [BBC+02, FD00] can manage leave of nodes, they cannot manage join of nodes or migration of node names. In contrast to these systems, our contribution is that we proposed the user interface that allows flexible adjustment of join/leave of nodes.

Secondly, they do not work well under common security polices. MPICH-G [FK98, KTF03] is a grid-enabled implementation of the MPI. MPICH-G allows a user to run MPI applications on multiple machines, potentially of different architectures. Moreover, its authentication mechanism enables a user to run MPI programs on resources distributed over multiple subnets. However, it needs to modify administrative restrictions to use MPICH-G under common security polices such as firewalls since its implementation assumes that any-to-any direct communication is allowed (e.g., no connections are blocked by firewalls or NAT). On the other hand, Phoenix automatically works around common security polices without modifying administrative restrictions. We believe Phoenix significantly decreases a user's cost for utilizing remote machines and provides smooth uti-

lization of computational resources.

MPICH-V [BBC+02] is an automatic volatility tolerant MPI based on uncoordinated checkpoint/rollback and distributed message logging. It provides fault-tolerance as well as communication mechanism that bypasses firewalls. Even if some of computation nodes are inside a firewall, every node can communicate with one another. This is because all the messages between computation nodes are relayed via Memory Channels (CMs), which are globally reachable. Whereas this mechanism is suitable for providing fault tolerance (e.g., message logging), it is less flexible than our communication mechanism for the following reasons. Firstly, CMs must be configured manually and must be stable. Secondly, computation nodes do not communicate with each other even if they can connect directly. In contrast to MPICH-V, Phoenix calculates forwarding routes automatically by routing algorithm.

Programming Models for Accommodating Dynamic Join and Leave of Nodes. Bayanihun [Sar99] proposed a programming models and interfaces for volunteer computing systems called Bulk Synchronous Parallel (BSP) model. BSP provides programmers with familiar message passing and remote memory primitives while remaining flexible enough to be used in dynamic environments. However, in Bayanihun, computation must be separated into a local computation phase and a global communication phase. Each task cannot send/receive messages in a local computation phase, and messages are exchanged only in a global communication phase. Since it requires all the tasks to be synchronized between two phases, it is not appropriate for general-purpose applications that involve frequent and irregular communications between nodes.

JXTA [VNRS02] is a set of generalized peer-to-peer protocols for a dynamic and decentralized organization of computational resources. In JXTA, each node communicates with one another as follows. When a node wants to send a message to another node, it first sends a query to some accessible routers, which collect routing information from the other routers. Then, the node sends the message to an appropriate node using the routing information obtained from the routers. JXTA allows nodes to bypass firewalls by relaying messages to a globally reachable nodes called relay peers. Since JXTA basically does not provide a method fpr configuring routers and relay peers automatically, it must be configured manually. As previously mentioned, such manual configuration is not feasible for networks where nodes join and leave dynamically.

Peer-to-Peer Information Sharing Systems. Peer-to-Peer information sharing systems such as Pastry [RD01], Tapestry [ZKJ01], Chord [SMK+01] and CAN [RFH+01] provide a distributed shared hash table. These systems are completely decentralized and self-organizing. It automatically adapts to arrival, departure, and failure of nodes. They proposed efficient routing algorithms to lookup/insert items in the hash table. For ex-

ample, Pastry assigns each nodes a unique ID. Then it routes a message to the node with a node ID that is a numerically closest to the destination address of the message. This algorithm notably reduces both the size of the routing table and the number of routing table update messages.

However, their algorithms assume that every node can communicate directly with one another and that nodes can always forward messages to appropriate nodes. Thus they cannot work on environments where some direct connections are blocked by security policies (e.g., firewalls).

2.6 Summary

We have described the Phoenix library, a wide-area message passing system for accommodating dynamically joining/leaving resources. Phoenix provides a collection of virtual node names that programmers dynamically allocates to or de-allocates from processes. With this dynamic allocation of virtual node names, the library supports nodes joining and leaving computation at any time. In addition, Phoenix supports message routing between nodes not directly reachable due to firewalls and/or NATs.

We evaluated the performance of the Phoenix library using several benchmark programs, including a parallel ray-tracing program based on Pov-Ray and Integer Sort in NAS Parallel Benchmark suite. The parallel ray-tracing that distributes workloads by a divide-and-conquer algorithm achieved a good speedup with a large number of nodes across multiple LANs (about 78 times speedup using 104 CPUs across three LANs).

Our future work is to make Phoenix more feasible for the deployment of wide-are applications on real network environments, including:

Efficient implementation of collective operations The Phoenix library currently provides no primitive for collective operations; a programmer need emulate collective operations (e.g., broadcast, multicast) with combining unicast operations as mentioned in Section 2.2.2.

The proposed mechanism for supporting collective operations by forwarding the message multiple times at the application level is clearly inappropriate. As existing message-passing infrastructures support broadcast (e.g., `MPI_Alltoall`), true broadcast must be supported.

Convenient Programming Interface In current implementation, Phoenix provides only the primitive interfaces for message passing. For example, it is too complex and cumbersome for a user to manually deal with migration of virtual nodes and application data structures in a deadlock-free manner [TKEY03].

Thus, we would like to provide more convenient programming interfaces that allow easy and efficient implementation of parallel applications, such as a shared object space like Linda [CG89] or divide-conquer systems like Clik-NOW [BL97].

Failure detection mechanism When Phoenix is unable to forward messages to a certain virtual node, it queues them for an unbounded period of time. This does not seem like a sufficient solution. Queuing many messages will induce a major overhead, first on storage space, and then on bandwidth once the missing nodes re-join.

The usual solution in the distributed computing world (e.g., in group communication systems) is to bound the time period in which messages are queued, and after a certain timeout, notify the application of the failure and discard the messages. The application will be able to process these tasks by itself or send them to others.

We should note that some of future work has been carried out by other members of our Phoenix project. Performance evaluation of other applications (e.g., LU factorization) is described in [ETKY04]. The implementation of efficient collective operations for WANs is described in [STC05]. A scalable fault detection mechanism is presented in [HTC05].

Chapter 3

Routing and Resource Discovery in Phoenix

Overview

We describe a communication subsystem of the Phoenix message passing library. It supports (1) message routing between nodes not directly reachable due to firewalls and/or NATs, (2) resource discovery facilitating ease of configuration that allows nodes without static names (e.g., DHCP nodes) to participate in computation without additional work, and (3) nodes dynamically joining/leaving computation during runtime. We argue that, in future Grid environments, all of the above functions, not just routing across firewalls, will become important issues of Grid-enabled message passing systems including MPI. Unlike solutions commonly proposed by previous work on a Grid-enabled MPI, our system employs a distributed resource discovery and routing table construction algorithm, rather than assuming all such pieces of information are available in a static configuration file or similar form. Experimental results using 400 nodes in three LANs indicate that our algorithm is able to dynamically discover participating peers, connect them, and calculate a routing table. The elapsed time of our algorithm is only about twice as long as that of offline route calculation that just connects nodes based on a fully given configuration.

3.1 Introduction

A message passing model is a dominant programming model for high performance parallel computation involving a large number of (e.g., > 100) nodes. It may be even more so in future multi-clusters and/or computational grids, where programmers carefully need to optimize communication of applications. Thus it is natural for researchers on HPC to seek a message passing library suitable for such environments.

There have been a great deal of work with this end, most of which aim at building "Grid-enabled" MPI libraries [FK98, KHB+99, ITKT00, BBC+02, AM03]. A primary design/implementation issue is how to deal with the fact that nodes may not be directly reachable in the underlying communication layer (e.g., TCP). This issue arises due to IP filtering as well as NAT/DHCP.

As far as we know, all existing systems essentially let a user specify the routes offline (e.g., in a configuration file). Most typically, a configuration file groups nodes and specifies a gateway node (either for each group or for the entire nodes) via which messages between directly unreachable nodes are routed. This solution is simple to implement and feasible for a small number of nodes distributed over a couple of clusters. The solution, however, must be generalized and extended in several ways for future environments, as discussed below.

One obvious issue is a scalability limitation due to gateways. A user should be able to specify as many gateways as permitted by a network administrator, rather than just one for each cluster. More important, having more resources spread over the Grid implies that resource selections tend to become more dynamic and adaptive. It will thus quickly become impractical for the user to maintain a *complete* resource description, which works for all possible set of resources that might be selected. Note that in the Grid setting, a complete description not only involves a list of resources, but also specifies routing (i.e., connectivity between nodes). Nodes that have dynamic IP addresses are more troublesome. Though they are able to participate in computation with a suitable resource manager support, it would be difficult for the MPI user even to specify such nodes in what would be called a "complete" configuration file.

All in all, neither routing nor the names of participating nodes should be completely specified by the user; communication libraries must learn them whatever resources are selected by the scheduler.

To this end, we have developed the Phoenix message passing library. This chapter describes design and implementation of its enhanced routing and peer discovery facility only briefly addressed in Chapter 2. Specifically, it allows nodes to connect each other without initially knowing all the peer names participating in computation. Then the nodes build a routing table according to the resulting graph of connections. The initial knowledge of the nodes is only the names of a small (arbitrary) number of "hub" nodes, through which nodes learn names of other participating nodes and bootstrap the entire connection graph.

The mechanism is implemented in a fully dynamic fashion, in that it allows nodes to join and leave at an arbitrary point of execution. Such a fully dynamic peer discovery and routing table construction is mandatory if a parallel programming model supports dynamic processes (e.g., Phoenix, Dyn-MPI [WLNL03], PVM [GBD+94], and MPI2 [Mes03]). In addition, the mechanism is a natural facility even if the model, *per se*, only supports static processes. This is because, as we have mentioned, dynamic and

adaptive resource schedulers, and/or even a very primitive form of fault tolerance (e.g., a mechanism that avoids initially dead nodes) makes selected resources not completely predictable by a user. Migration of MPI jobs and fault-tolerant MPIs [FD00, BNC+01, BBC+02] also need such mechanism since it enables nodes and networks to change dynamically.

Technically, our system consists of routing table construction and resource discovery. Thus, we borrowed basic ideas from a body of work on routing [RT99] and resource discovery [HBLL99, KPV01, KP02, AD03]. Specifically, our routing table construction algorithm is based on the Destination Sequenced Distance Vector (DSDV) routing algorithm [PB94], originally proposed in the context of mobile ad-hoc network routing. Our experiments indicated, however, that naively adopting the algorithm for our purpose does not scale because, in our setting, the connection graph is dense and/or the connection graph sometimes changes very rapidly (e.g., at start up). By carefully engineering propagation and scheduling of routing events, we dramatically improved its performance. We also show this is achieved even when nodes initially know only a small number of other processes. That is, resource discovery does not affect the performance.

The remainder of this chapter is organized as follows. Section 3.2 reviews existing Grid-enabled MPIs. Section 3.3 describes our problem setting. Section 3.4 gives the details of the routing and resource discovery algorithm. Section 3.5 presents experimental results. Section 3.6 discusses related work. The final section summarizes the chapter.

3.2 Grid-Enabled MPIs

3.2.1 Requirements

We summarize requirements on Grid-enabled communication systems , especially focusing on MPI. First, message forwarding routes must be shortest. For high performance communication, nodes must be able to transmit messages directly if possible. Second, they must work on various network topologies where many restrictions are imposed. Third, in many contexts, it is highly desirable for them to allow dynamic changes of the connection topology. This is true even if the computation model only allows a static number of processes. For example, many fault tolerant MPIs such as MPI/FT [BNC+01] have been developed. In these systems, crashed processes may be restarted on different machines. There are also systems that balance system loads adaptively (e.g., DynMPI [WLNL03], dynamic load balancing on LAM/MPI [MSK03]). These systems dynamically change the allocation of MPI ranks or the number of nodes that participate in computation. To support such systems, the routing mechanism must adapt to dynamic changes of nodes and connections.

3.2.2 Existing Systems

MPICH-V [BBC+02] is an automatic volatility tolerant MPI based on uncoordinated checkpoint/rollback and distributed message logging. It provides fault-tolerance as well as a communication mechanism that enables nodes to communicate across firewalls. To bypass firewalls, MPICH-V prepares Channel Memories (CMs), which must be globally reachable from all the nodes. The system enables nodes to communicate with one another by relaying messages via CMs. This indirect communication of MPICH-V has three drawbacks. First, every node always needs to communicate with each other via CMs even if they can communicate with each other directly. Second, the network topology that MPICH-V supports is limited. It requires as least one globally reachable node in networks. Third, it cannot tolerate dynamic changes on network topology since CMs are assumed to be fixed.

Stampi [ITKT00] and PACX-MPI [GRBK98] provide unified MPI interfaces for heterogeneous networks. They can exploit multiple clusters that may belong to different private networks. Stampi creates a message routing process that relays messages between them when machines in different clusters cannot communicate directly with each other through IP. PACK-MPI provides a similar facility by having proxies that handle inter-cluster communication. They cannot tolerate dynamic changes to connection topologies. In addition, they support only limited connection topologies; routing processes/proxies must be globally reachable.

MPICH/MADIII [AM03] offers a forwarding mechanism for inter-cluster communication. It automatically calculates forwarding routes for every machine using manually given information about the entire network. Since the forwarding routes are calculated statically at start up, MPICH/MADIII cannot tolerate dynamic changes of the connection topology.

3.3 Problem Setting

We assume the system assigns each participating node one or more *application level* names, or simply *logical* names. Node rank in MPI is an example of a logical name. The programmer uses logical names to specify message destinations. In contrast, a *physical* name refers to a name used to communicate in the underlying communication layer. For example, if we build MPI on top of TCP, a physical name is a pair $\langle$*hostname*, *port number*$\rangle$. The basic job of the communication library is to route messages with their destinations specified by logical names to the right destination node, even though it may not be directly reachable in the underlying communication layer.

As we mentioned in the introduction, we generalize the problem as follows. First, the communication library allows network connectivity to change at runtime. It changes

routing accordingly. Second, it allows nodes to be added at runtime without initially knowing their (either physical or logical) names. When a new node joins a computation, the new node must know, of course, at least one physical name of an already participating node. On the other hand, any participating node does not have to know the new node in advance. Nodes may also be deleted at runtime. As mentioned previously, they are slight generalization of a minimally dynamic process model where resources are selected by the scheduler at a job start up depending on the availability and loads of resources.

Although our system has been implemented in the context of Phoenix message passing model [TKEY03], none of the algorithms described in this chapter depend on the specifics of Phoenix model.

3.4 Routing and Resource Discovery Algorithm

3.4.1 Overview

We briefly sketch the behaviour of our algorithm using an example network illustrated in Figure 3.1. It consists of two subnets X and Y. Node a and e are gateways of X and Y respectively, and they have fixed names. Node b, c, and d are configured with DHCP in subnet X. Node f and g are also DHCP clients in subnet Y. They do not have any static names. Firewalls are installed on both subnets. They block connections between non-gateways belonging to different subnets. The only allowed connection across the firewalls is SSH [Opeb] connection (port 22) between a and e.

The system roughly works as follows.

Step 1: Initial setup A set of processes bring up on resources chosen by a scheduler or a user. Each process knows physical names of *some* (not necessarily all) participating nodes. Let us assume in Figure 3.1, nodes only know the physical names of the two gateway nodes. This is a small piece of configuration information comfortably kept in each node or even passed upon command submission. The system currently supports three underlying communication protocols: direct TCP, OpenSSL [Oped], and SSH tunneling. SSH is useful in many cases where the only inbound connection allowed is SSH port (22).

Step 2: Overlay network construction Each node tries to establish connections to machines it knows. In Figure 3.1, the DHCP clients will succeed in establishing direct TCP connections to one of the gateways. The gateways also establish SSH connections between them.

Step 3: Resource discovery and routing table construction Each node constructs its routing table by exchanging messages on the overlay network. By looking up the routing table, each node determines which neighbor a message should be transmitted

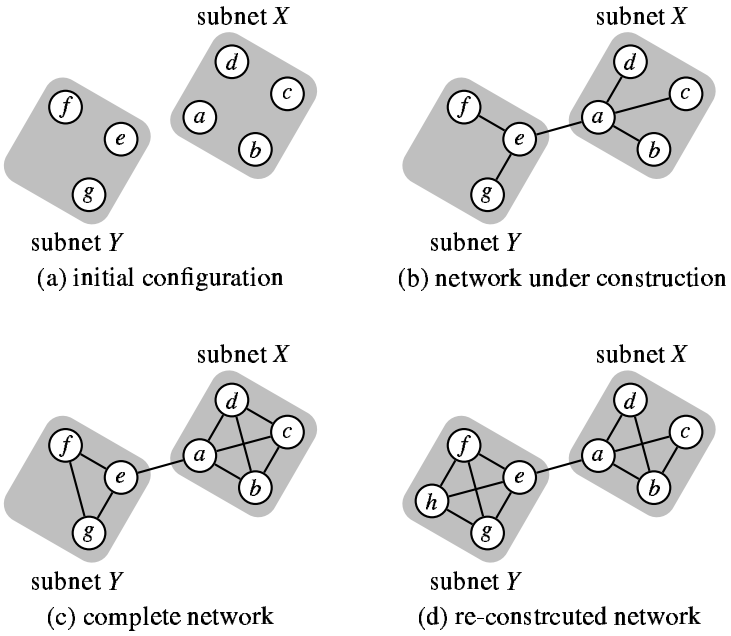

Figure 3.1: Process of route calculation. Nodes and solid lines respectively indicate machines and established connections between machines. (a) shows initial configurations. (b) shows an overlay network constructed only by the initial configurations. (c) shows an overlay network completely constructed when the system stabilizes. (d) shows a reconstructed network when h is added to the network.

to. On the receipt of messages destined for other nodes, a node also looks up the table to determine a forwarding route.

Each node also learns new machines it initially does not know from messages it receives. When a node finds physical names it does not know, it retraces **step 2** and **step 3**. The steps are repeated until each node knows all the participating machines and all the possible connections are established. This mechanism can minimize the number of hops each message travels. For example, in Figure 3.1, the DHCP clients in the same subnet can eventually communicate with each other directly even if they initially do not know each other.

The system guarantees that each node eventually knows all the available machines if the graph is connected after the first execution of **step 2**.

Note that **step 2** and **step 3** may interleave. Thus the system can route messages before the routing table is fully stabilized. Whenever the connection topology changes, the overlay network and the routing tables are updated. For example, suppose machine h is added to subnet Y, and h initially knows e. In this case, the overlay network is reconstructed and finally h becomes directly connected to e, f, and g.

3.4.2 Destination-Sequenced Distance-Vector Routing

Our routing algorithm is based on Destination-Sequenced Distance-Vector Routing (DSDV) algorithm [PB94] proposed for mobile ad-hoc networks. It gives us a good starting point because it adapts to changes of the connection topology and consumes a relatively small amount of memory compared to other schemes based on distance-vector. In DSDV, each routing table, at each node, lists all available destinations. Specifically, the entry for destination node v consists of:

- fwd: a node to which messages destined for v are forwarded.
- $nhops$: the number of hops of the route from the local node to destination v.
- seq: a sequence number that implies the freshness of the entry, as will be explained later.

Hereafter $R_u[v]$ is used to denote the entry for destination node v in u's routing table. $R_u[v].fwd$, $R_u[v].seq$, and $R_u[v].seq$ are used to describe fwd, $nhops$, and seq of $R_u[v]$ respectively.

To maintain the consistency of the routing table in dynamically varying topology, each node transmits (a subset of) its routing table to update its neighbor's routing table. Each node basically broadcasts when its routing table is updated by significant new information (e.g., discovery of a shorter path, break of a connection). On the receipt of a

message, each node updates its routing table by the following rules: routes with larger sequence numbers are always preferred as the basis for making forwarding decisions; and of the path with the same sequence number, shorter routes are chosen.

To calculate the shortest paths without any loops, the sequence number is maintained in such a way that the most recently updated entry has the largest sequence number among all nodes. For example, when a node finds a broken link, the entries of which route depends on the broken link become obsolete. In such a case, the node broadcasts these entries with incrementing their sequence number to update the other nodes' routing tables correctly.

Note that the receipt of update messages may cause another transmission of update messages to make the routing table of all the nodes consistent. The message transmission is repeated until all the nodes in the network have received a copy of the update message with a corresponding metric.

3.4.3 Resource Discovery Algorithm

As described in Section 3.4.1, each node needs to discover available machines that it does not know in the beginning. Each node needs to collect information about available machines by exchanging messages with other nodes.

The node discovery is performed as follows. Initially each node only knows a part of machines participating in the application. When a node transmits a routing table message to update u's entry, it attaches u's physical name. On the receipt of this message, the receiver learns u's physical name, and tries to establish a connection to it.

3.4.4 Performance of the Naive DSDV

As we will show in section 3.5, performance of a naively implemented DSDV is just poor when the number of nodes becomes large (> 50). We investigated this and found there are two primary reasons.

- Sending routing update messages to every neighbor result in many useless or redundant messages when the network is dense, as is usually the case in our problem setting.
- When the application brings up or when many nodes are simultaneously added to the application, the set of known node names as well as the topology of the graph change very rapidly. Naively running DSDV update protocol per each small change in the graph turns out to be a very inefficient way of calculating the final routing table, as we will see below.

For the first bullet, the topology of the overlay network in our problem setting is typically dense since many nodes can, and would like to, directly communicate with

each another via the underlying communication layer. For example, the topology of the network consisting of multiple clusters is usually a collection of cliques. These dense networks cause a large number of redundant message transmissions. Suppose that one node updates its routing table. The minimum number of messages required to update all the routing tables is $N - 1$ where N is the current number of nodes. This is because a message must be transmitted to each node at least once to deliver new information. On the other hand, the number of the messages that are transmitted until the naive DSDV stabilizes is $O(E)$ where E is the number of edges. This is because DSDV propagates update messages via all the edges. In dense networks, E is much larger than N (e.g., $E = \Omega(N^2)$). Thus the number of exchanged messages becomes large compared to the minimum $N - 1$. We should point out this will not be a big issue in the context of mobile ad-hoc networks because the networks are typically sparse; neighbors of a node are limited to those physically close to the node. In contrast, connections are established between every allowed pairs of nodes in our problem setting.

For the second bullet, consider what will happen when many nodes are simultaneously added to the network (or when an application brings up). When a node accepts a connection from a new node or receives an update from a neighbor, it updates its routing table and sends update messages to neighbors. In general, this must be done promptly to propagate the new piece of information as fast as possible. When many nodes join an application almost simultaneously, however, updating neighbor nodes too eagerly result in many small messages that could have been merged when we know there will be subsequent updates.

3.4.5 Optimizations

We optimize the naive DSDV based on the two observations discussed above.

Eliminating Redundant Updates. Suppose that node u transmits an update messages to its neighbors N_u. The algorithm guarantees that nodes in N_u do not propagate the update message to each other, as they get it from u anyways.

This is implemented simply by adding two fields $trans$ and $rcpt$, to each entry of the routing table. $R_u[v].trans$ is the collection of nodes to which u has already transmitted $R_u[v]$, whereas $R_u[v].rcpt$ the collection of nodes that received or will soon receive the entry corresponding to $R_u[v]$ from some node [1].

Thus $w \in R_u[v].trans \cup R_u[v].rcpt$ indicates that u does not need to transmit a message to w; w has already received $R_u[v]$ or will soon receive $R_u[v]$. Node u must transmit $R_u[v]$ to u's neighbor w only if w does not belong to $R_u[v].trans \cup R_u[v].rcpt$.

[1] To identify each node uniquely, we basically use an IP address. When IP addresses are not unique among nodes, random bits are added to each node's identifier.

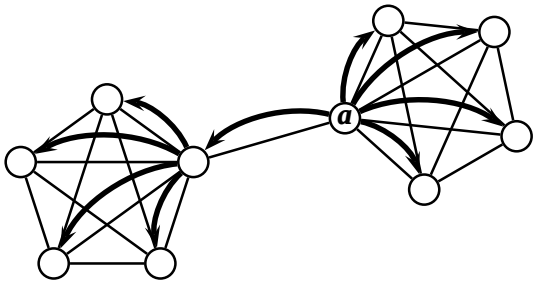

Figure 3.2: An example of elimination of redundant message transmission

Fields $trans$ and $rcpt$ are maintained as follows. When u sends $R_u[v]$ to a set of nodes V, V is added to $R_u[v].trans$. When w receives an entry e from u, $R_w[v]$ is updated as follows. If either $R_w[v].fwd$, $R_w[v].nhops$, or $R_w[v].seq$ is updated by this message, $R_w[v].trans$ becomes an empty set and $R_w[v].rcpt$ becomes a singleton that only contains sender u. Otherwise sender u and $e.trans$ are added to both $R_w[v].trans$ and $R_w[v].rcpt$.

Figure 3.2 shows an example of the elimination of redundant message transmission. Let us consider entries for destination v. Suppose that for all u both $R_u[v].trans$ and $R_u[v].rcpt$ are initially empty. Then a broadcasts $R_a[v]$, which is freshest among all the nodes. As the arrows in Figure 3.2 indicate, only a needs to broadcast update messages to make all the nodes' entry fresh.

Merging Clustered Updates. Our second optimization tries to merge messages for many routing table updates that occur almost simultaneously. Simply buffering messages for a fixed period would sacrifice performance when an update occurs in isolation. Thus we address the problem by the following scheduling policy of events related to routing.

1. If a node knows another node but does not have a connection to it, it connects to the node with the highest priority.

2. If a node connects to all its acquaintance, but has some unprocessed messages updating its local routing table, it processes these messages.

3. Otherwise, it sends update messages to its neighbors.

In short, we give priorities to routing-related events in the following order. (1) making connections, (2) updating the local table, and (3) propagating updates to the neighbors' tables. Here, all updates that have not been propagated to a neighbor are merged into a single message.

Table 3.1: Experimental environments

	CPU	**# of CPUs**
subnet A	UltraSPARCIII 750MHz	2CPU x 112 nodes
subnet B	Xeon 2.40GHz	2CPU x 64 nodes
subnet C	PentiumIII 800MHz	2CPU x 16 nodes
	PentiumIII 1.4GHz	1CPU x 16 nodes

3.5 Experiments

Table 3.1 summarizes the experimental environment. Machines in the same subnet can communicate directly with each other. The inter-subnet communication is restricted: each subnet has a gateway, which is the only machine that can accept inbound connections, at SSH port (22).

First, we measured the elapsed time of routing table construction without node discovery. That is, we give all node names to each node offline (via a configuration file). We conducted the experiment on a single subnet (A) and the three subnets.

To begin with, let us confirm that the naive DSDV performs poorly, as shown in Figure 3.3 and Figure 3.4. It does not scale at all when the number of processes become > 50. Thus we removed it from further investigation. Our interest is the price we pay for supporting the general, fully dynamic process model. So we compared our algorithm with two "easier" cases where processes are assumed to be static, or the process configuration is completely given offline.

Figure 3.5 shows the result. The upper graph is for the single subnet experiment and the lower graph for the three subnets case. For the latter, we proportionally mixed CPUs from the three subnets [2]. The curve labeled "offline" is the simplest and the easiest setting. A configuration file describes a complete process configuration (which process should connect to which) and processes simply follow it. Thus, the elapsed time is mostly of just establishing connections. The curve labeled "master" assumes processes are static and their names known to every process, but connectivities between nodes are not known. It also assumes there is a master node and every process knows the path to the master node. Under this assumption, each process tries to connect to all other processes, learns its neighbors, and sends their names to the master. The master collects the messages and then calculates the all-to-all shortest paths. It finally sends the result to all processors.

When the number of processors is less than 100, all three cases have the approximately equal elapsed time. That is, our dynamic routing table construction has almost no overhead. Up to 400 processors, the elapsed time of our algorithm is within a factor

[2] UltraSPARCIII 750MHz : Xeon 2.40GHz : PentiumIII 800MHz : PentiumIII 1.4GHz $= 14 : 8 : 2 : 1$

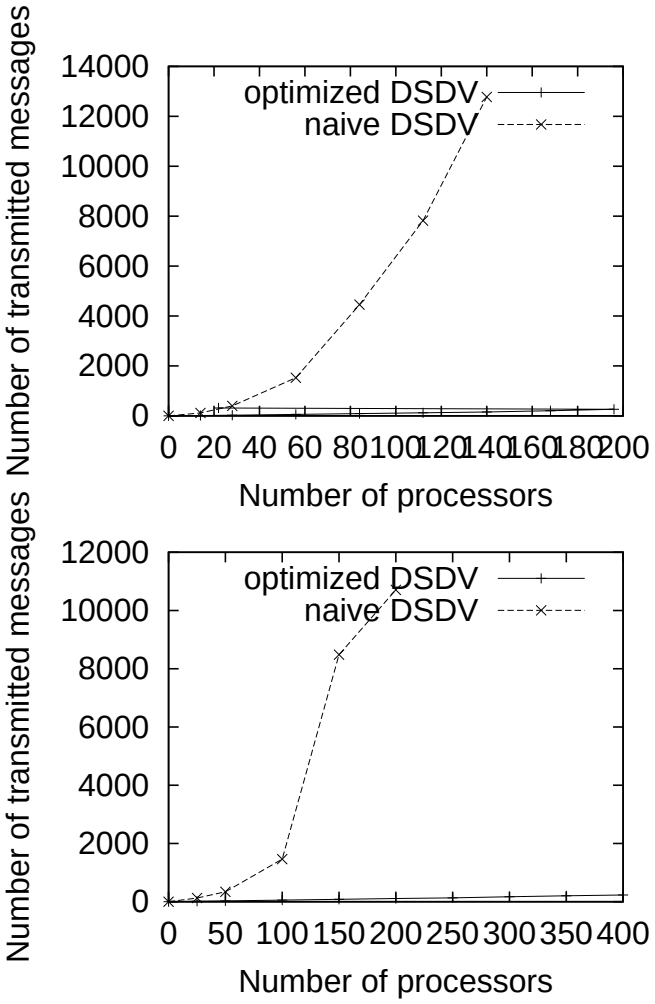

Figure 3.3: Number of transmitted messages of Naive DSDV compared with our optimized DSDV in a single subnet (upper) and three subnets (lower)

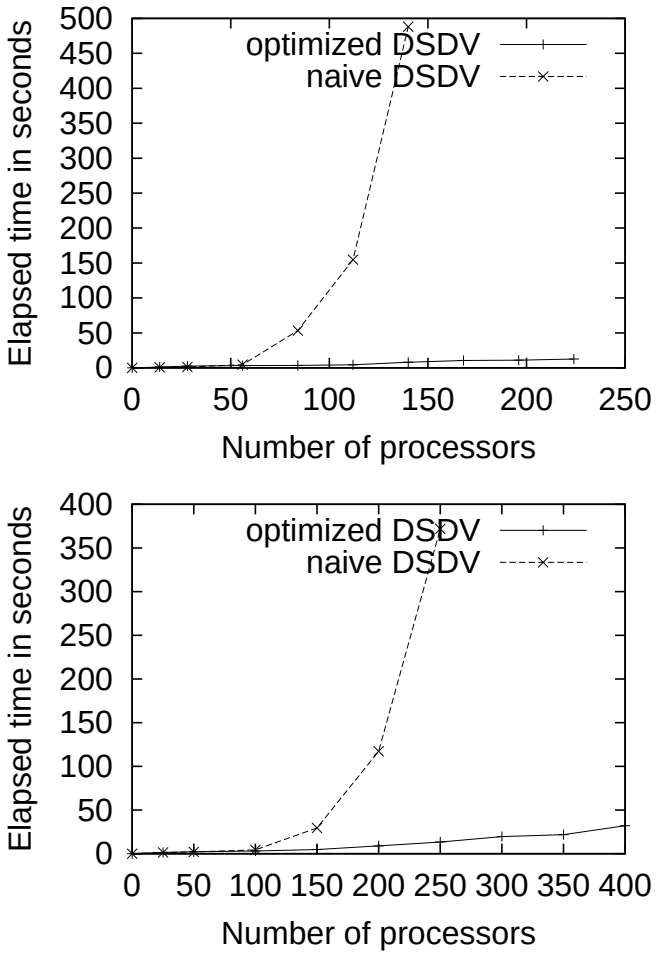

Figure 3.4: Time of Naive DSDV compared with our optimized DSDV in a single subnet (upper) and three subnets (lower)

of 2.3 of the offline case, and 1.5 of the master case.

Recall that our routing algorithm is fully dynamic, which means message send/receive can take place before the routing table is completely stabilized. Figure 3.6 gives us a sense of how much of the routes become "ready" at which point of calculation. The upper graph shows how much node pairs out of all the possible N^2 pairs are reachable at each moment (either directly or indirectly). All nodes are in a single cluster. As we can see, although it took 14 seconds to completely stabilize the routing table, more than 90% of node pairs become reachable at 10 second. The lower graph shows the average number of hops between reachable pairs at each moment. It should converge to one, and we almost get there at 10 second. In summary, it is fair to say most of the work has been done much earlier than the completion of the routing table construction.

Figure 3.7 shows how routes become ready on a dynamically changing network. In this experiment, we began with 224 processes and killed a half of the processes after 30 seconds have passed. Then after another 5 seconds have passed, we restarted the killed 112 processes. The result shows that the routing tables can be updated rapidly according to the addition/deletion of processes.

Finally, Figure 3.8 compares cases with or without node discovery. The result shows that the overhead of node discovery is negligible.

3.6 Related Work

Peer-to-Peer Information Sharing Systems. Peer-to-Peer information sharing systems such as Pastry [RD01], Tapestry [ZKJ01], Chord [SMK+01] and CAN [RFH+01] provide a distributed shared hash table. These systems are completely decentralized and self-organizing: each node automatically adapts to arrival, departure, and failure of nodes. They proposed efficient routing algorithms for looking up and inserting items in the hash table. For example, Pastry assigns each node a unique ID. Then it routes a message to the node with a node ID that is a numerically closest to the destination address of the message. This algorithm notably reduces the size of the routing table and the number of routing table update messages.

However, their algorithms assume that every node can communicate directly with one another and that nodes can always forward messages to appropriate nodes. Thus they cannot work on environments where direct communication may be prohibited by security policies (e.g., firewalls). In addition, these systems require forwarding messages via multiple nodes (e.g., in Pastry, $O(\log N)$ hops where N is the number of nodes in its network) even if nodes can communicate with one another directly. Since this unnecessary forwarding degrades communication performance heavily, their algorithms are not feasible for high-performance computing that involves dense communication.

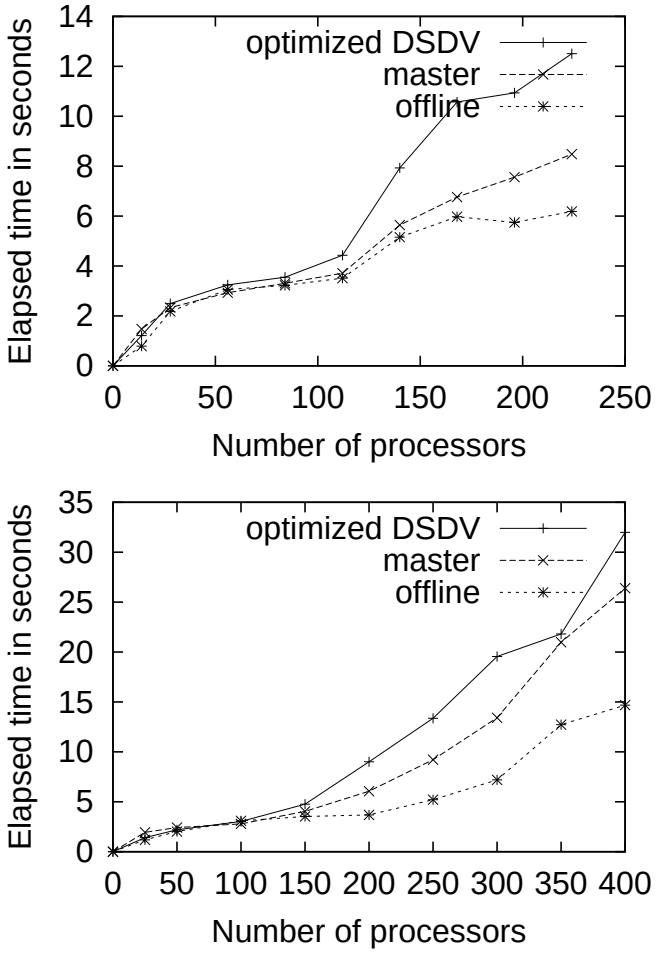

Figure 3.5: Comparison of our fully dynamic DSDV with two static cases (see text) in a single subnet (upper) and three subnets (lower)

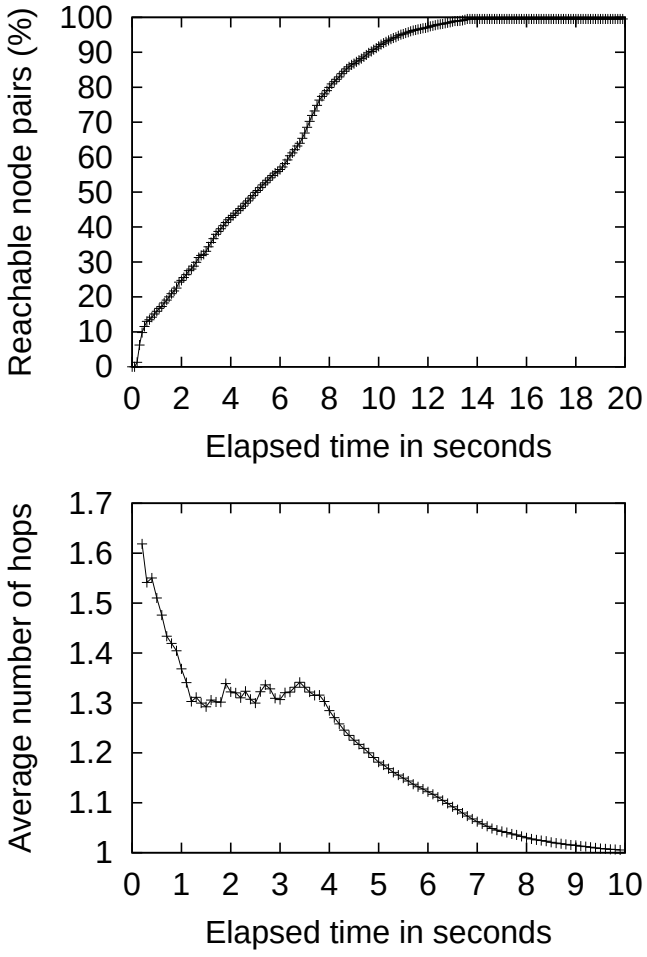

Figure 3.6: The fraction of reachable node pairs (upper) and the average number of hops between reachable node (lower)

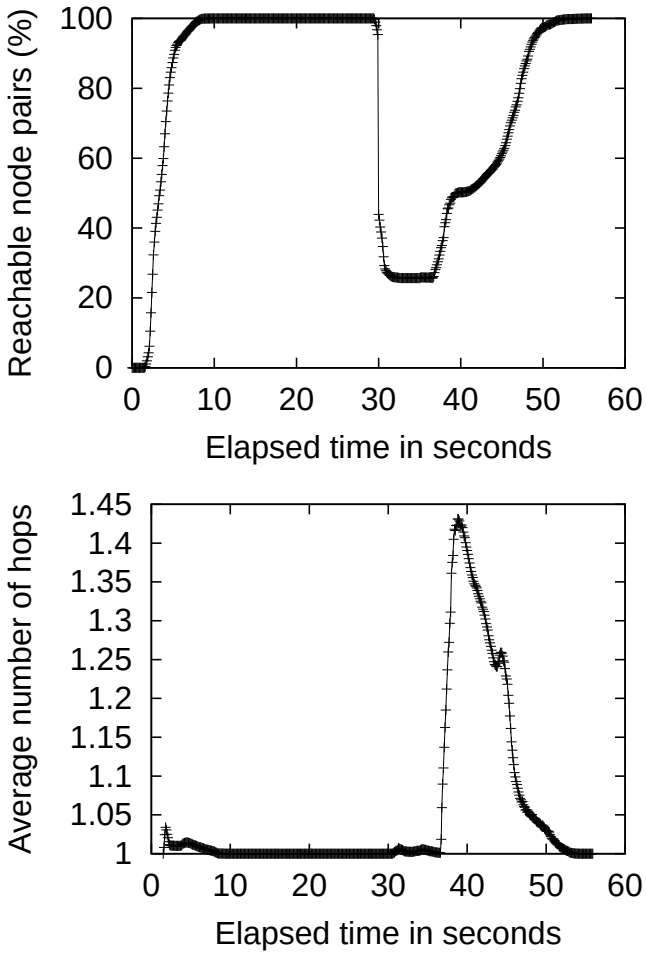

Figure 3.7: The fraction of reachable node pairs (upper) and the average number of hops between reachable node (lower) on a dynamically changing network

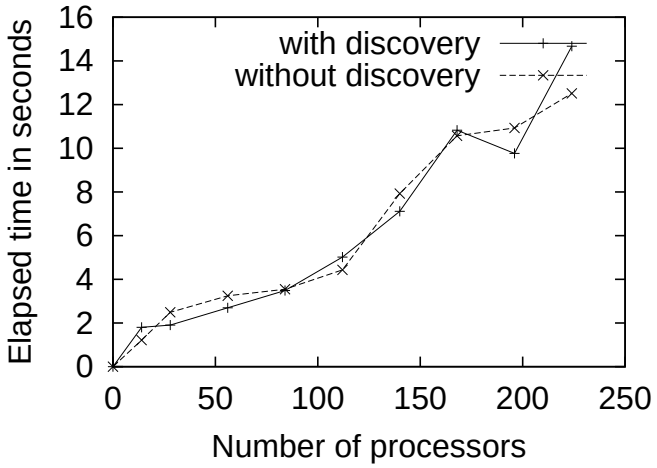

Figure 3.8: The elapsed time of the routing table construction with/without node discovery

Resource Discovery. A resource discovery problem introduced by Harchol-Balter, Leighton and Lewin in [HBLL99] is relevant for our algorithm. The resource discovery problem is to efficiently discover all the nodes that currently exist in the systems when each node initially knows only a small number of nodes. Though several algorithms for the resource discovery problems are proposed [KPV01, KP02, AD03], they do not suffice for our system. This is because they focus on only discovering node names and do not care routing.

3.7 Summary

We have described a communication subsystem for message passing systems for the Grid. It provides routing and resource discovery that tolerate dynamic changes of connection topologies. We evaluated the performance of the algorithm by running the system on 400 nodes in three LANs. When the number of processors is less than 100, our dynamic routing table construction adds almost no overhead to the case where the network connectivity is completely given offline. In all cases, the elapsed time of our algorithm is within a factor of 2.3 of the offline case. Furthermore, 90% of node pairs become reachable much earlier than completion and messages can be routed when the routing table is being constructed.

Improvement of scalability We plan to improve the scalability of Phoenix to utilize emerging computational grids approaching thousands of nodes (e.g., Grid'5000 project [CDD+05]).

Possible optimization techniques include compaction of a routing table, reduction of the number of routing update messages, and efficient connection management that considers underlying physical networks. Although much work on optimization techniques of routing algorithms has been carried out both in theoretical and practical research fields (e.g., compact routing [TZ01]), most of these algorithms have not yet been applied to computational grids. Further studies must to be conducted to reason that the algorithms can be applied to Phoenix.

Support of reliable data transmission The current implementation assumes that no message is lost by link failure and that links are never broken while the system is sending messages. Obviously, this assumption is not feasible for wide-area networks where link failure occurs frequently.

In addition, the current implementation does not guarantee that the order of delivered messages is preserved when the topology of networks changes dynamically.

To solve the above limitations, we plan to implement a message re-transmission mechanism on the application layer.

Chapter 4

Virtual Private Grid: A Command Shell for Utilizing Hundreds of Machines Efficiently

Overview

We describe design and implementation of *Virtual Private Grid (VPG)*, a shell that utilizes many machines distributed over multiple subnets easily and efficiently. VPG works around common security policies (e.g., firewall, private IP, DHCP) that restrict communication between machines and even break uniqueness of IP addresses. VPG provides the following functions. (1) A unique nickname to each machine that does not depend on a DNS name or a fixed IP address. (2) Job submissions to any nicknamed machine. (3) Redirections from/to a file on any nicknamed machine. (4) Network pipes between commands executed on any nicknamed machine. We present two ways for implementing the communication mechanism of VPG on dynamic environments. The first method is based on the Phoenix library, and the second method on construction of self-stabilizing spanning tree. We ran VPG on about 100 nodes (270 CPUs) to demonstrate its feasibility. We measured a turn around time of a small job submission with VPG and other tools: rsh, SSH, and globus-job-run (a remote job submission tool provided by Globus []). The experimental result shows that VPG can submit a job faster than SSH and `globus-job-run` since VPG performs authentication only when it constructs a tree.

4.1 Introduction

Today, computer users commonly have an access to hundreds of machines across multiple subnets and geographically distributed places. In such environments, they can potentially archive high performance for certain types of parallel applications (e.g., parameter sweep applications). Job submission tools, such as Condor [FTF+01] and Portable Batch System (PBS) [Opea], enable a user to submit many jobs to clusters/supercomputers in a local network.

However, when available machines are distributed over multiple subnets, it becomes difficult and cumbersome to utilize them with these tools. We explain this problem below.

Machines distributed over multiple subnets are usually managed by different administrators, who impose various restrictions on their use for the sake of security and ease of administration. Examples of these restrictions are,

Firewall A firewall protects local machines from malicious attacks by restricting accesses from external machines. For example, IP filtering—a typical firewall configuration—restricts connections from/to machines with a particular IP address.

Private IP A private IP is an IP address that is visible only within a subnet. Machines outside a subnet cannot establish direct connections to machines that have only a private IP address. In addition, since a private IP address is visible only within a subnet, machines in different subnets may have the same private IP address without confusion. This breaks the uniqueness of IP addresses.

DHCP client DHCP is a mechanism that enables machines to extract their network configuration from a DHCP server. An IP address of a DHCP client changes dynamically whenever it extracts a new configuration (typically when it reboots).

Because of the above restrictions, machines cannot necessarily establish a direct connection to every other machine or may even not have unique addresses. Hence, a user may not submit a job easily across multiple subnets with existing tools; s/he has to work around the restrictions with ad-hoc methods, which are found on a case-by-case basis with human intervention. For example, when submitting a job to machines behind a firewall, a user usually has to first log onto a gateway machine and then onto the target. Accessing a DHCP client requires some database that stores its IP address. A situation becomes more complicated if those addresses are private IP addresses.

In addition, we must consider that the number of available machines is large (e.g., > 100) and that the topology of the network usually changes dynamically (e.g., machines may crash or the network may become disconnected). These also make it difficult for a user to manually work around the administrative restrictions. For instance, a user has to know which machines are available at present; and s/he may need to find a new job forwarding route whenever a machine crashes.

To summarize, the administrative restrictions significantly increase the user's cost for utilizing remote machines, and consequently, obstruct smooth utilization of computational resources. A user would like to have a solution in which all machines can be reached directly and transparently, with names fixed over time.

To this end, we have developed *Virtual Private Grid (VPG)*, a command shell that can utilize hundreds of machines efficiently. It enables a user to easily submit jobs to remote machines by providing the following mechanisms.

Nickname mechanism It gives each machine a unique name that does not depend on a DNS name or a fixed IP address.

Communication mechanism A user can directly access remote machines that would be reachable by using existing tools (e.g., rsh, SSH [Opeb]) several times from her/his local machine. A user can access these machines merely by specifying them with their nickname.

The above mechanisms can be implemented without modifying administrative policies and can tolerate dynamic changes of the topology of the network, though some manual configurations are required. We implemented VPG in two ways: implementation based on a self-stabilizing spanning tree construction [AKY91, AK93] and based on the Phoenix library.

We ran the original implementation of VPG on about 100 nodes (270 CPUs) to demonstrate its feasibility. We measured a turn around time of a small job submission with VPG and other tools: rsh, SSH, and globus-job-run (a remote job submission tool provided by Globus []). The experimental result shows that VPG can submit a job faster than SSH and `globus-job-run` since VPG performs authentication only when it constructs a tree.

The remainder of this chapter is organized as follows. Section 4.2 shows difficulties in working around administrative restrictions through a practical motivating scenario. Section 4.3 describes the user interface of VPG. Section 4.4 and Section 4.5 present the details of the implementation of inter-process communication. Section 4.4 and Section 4.5 explain the implementation based on spanning tree construction and Phoenix respectively. Section 4.6 discusses limitations of VPG and solutions for overcoming these limitations. Section 4.7 shows experimental results. Section 4.8 mentions related work. The final section summarizes the chapter and states future work.

4.2 A Motivating Scenario

In this section, we show the difficulty of working around administrative restrictions through a practical motivating scenario. Consider the network shown in Figure 4.1. *Harp*, *tuba*, ... in Figure 4.1 represent host names. This network consists of three subnets including a DHCP client and a machine that has only a private IP. Firewalls restrict connections

between different subnets; SSH to the gateway machines (*harp*, *cscl0*, and *ise0*) is the only allowed in-bound connection. Such a configuration is fairly typical.

Suppose a user would like to submit jobs (commands) from *tuba* to all the other machines in the network. She/He needs to do the following depending on

- job submission from outside firewall to the inside. A user must typically first log onto a gateway machine ((i) in Figure 4.1).
- job submission to a machine that has only a private IP address. Similarly to the above, accessing such a machine requires first entering the subnet of the target machine ((ii) in Figure 4.1).
- job submission to a DHCP client. A user must somehow obtain the current IP address of the machine ((iii) in Figure 4.1).

If the number of machines is small, users may be able to perform the above steps manually. They may remember gateway machines to each machine and keep track of current addresses of DHCP clients. However, such ad-hoc solutions obviously do not scale to a large number of machines and subnets. Thus, we require a mechanism to access all the available machines directly, with names fixed over time.

One way to implement such a mechanism is to connect all the machines with bi-directional (e.g., TCP) connections and to route messages (e.g., job submission requests) via these connections. This enables a user to submit a job easily across multiple subnets. For example, even if a firewall blocks in-bound connections, a user can submit a job from outside firewall to the inside. She/He can forward the job via the connections that machines behind the firewall have initiated.

4.3 Design

4.3.1 User Interfaces

The following summarizes the functions provided by VPG.

- It gives each machine a (per-user) unique name that does not depend on a DNS name or a fixed IP address (*nicknaming*).
- It provides a job submission to any nicknamed machine.
- It provides a redirection from/to a file on any nicknamed machine.
- It provides a pipe between commands executed on any nicknamed machine.

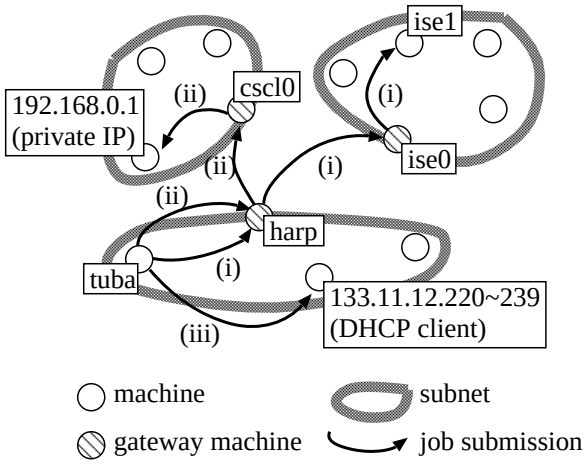

Figure 4.1: A practical example of a network on which various administrative restrictions are imposed

```
path@nickname
path@nickname > file@nickname
path@nickname < file@nickname
path@nickname | path@nickname
```

Figure 4.2: Shell syntax of remote job submission, redirection, and pipe

We make the above functions accessible by a combination of the simple shell syntax and existing commands (See Figure 4.2). For example, a user can submit a command to any remote machine by adding `@` followed by its nickname. A pipe can connect standard input/output of a program to output/input of another program running on a different machine.

The above functions require no modifications to existing administrative policies, and adapt to dynamic changes in the network conditions.

4.3.2 An Example of Job Submission

We show several examples of remote job submissions with VPG. Assume the same network as Figure 4.1 and that *tuba* is the home host, which a user initially logged in.

Figure 4.3 shows the machines' nicknames and the tree constructed by VPG. *A*, *B*, . . ., and *G* in Figure 4.3 represent nicknames, which do not change during execution of commands. For example, the DHCP client and the machine with a private IP in Figure 4.1 have unique and fixed nicknames *B* and *G* respectively. A dashed arrow from machine

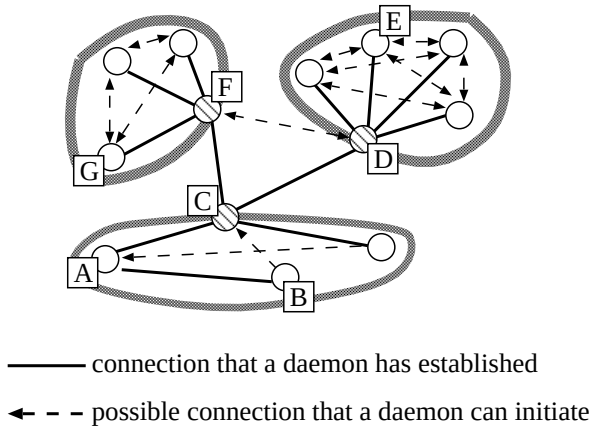

Figure 4.3: Job submission example

u to v is a possible connection that u can initiate to v. A solid line between u and v is a bi-directional connection that u and v have established between them.

Then, the user inputs the following command lines on the home host (i.e., host *A*).

`ps@B`
: This command line submits **`ps`** command to a DHCP client *B*. Note that the user can specify the target host by its nickname, rather than its IP addresses.

`tar@E -c file | tar@G x`
: This command archives **`file`** on *E*, transfers it to *G*, and extracts it on *G*. Note that *A*, *E*, and *G* belong to different subnets. VPG automatically detects a forwarding route ($A \rightarrow C \rightarrow D \rightarrow E$), submits the first **`tar`** command to *E* through the route, and executes it on *E*. Similarly, the second **`tar`** command is submitted from A to *G* (through the route $A \rightarrow C \rightarrow F \rightarrow G$), and the output of the first command is transferred from *E* to *G* (through the route $E \rightarrow D \rightarrow C \rightarrow F \rightarrow G$). As in UNIX pipes, the two commands are executed in parallel.

4.4 Implementation Based on Spanning Tree Construction

We describe the implementation of the inter-process communication mechanism of VPG based on a spanning-tree construction [AKY91, AK93]. VPG builds a spanning tree without knowing the topology of the whole network, and re-constructs it whenever the topology changes. This mechanism allows VPG to keep available machines connected with the

minimal number of connections, even if machines occasionally crash or networks become disconnected.

First, we formalize administrative restrictions. Next, we mention the self-stabilizing spanning tree algorithm [AKY91, AK93]. With this algorithm, VPG daemons select necessary connections to make all the machines available. Then, we present the algorithm that calculates routes to participating machines for job submissions, redirections, and so on.

4.4.1 Network Model

We model the configuration of the network as a directed graph $G = (V, E)$, where V is the set of machines, and E the set of possible (i.e., allowed) connections. An edge is labeled either 'regular' or 'ssh'. Let u and v be machines. If (u, v) is in E and labeled 'regular', u can initiate a regular connection to v. If (u, v) is in E and labeled 'ssh', u can initiate an SSH connection to v. In the following, labels are omitted when not important.

In this framework, administrative restrictions are modeled as follows.

Firewall For example, if a firewall blocks all in-bound connections to a subnet, we have

$$\begin{aligned}] \quad & \forall u \in \{\text{machines outside the subnet}\}, \\ & \forall v \in \{\text{machines inside the subnet}\}, \\ & (u, v) \notin E \end{aligned}$$

Let g be a gateway reachable via SSH from any machine. Then we have

$$\forall u \in V,\ (u, g)_{ssh} \in E$$

Private IP If u is a machine with a private IP address, we have

$$\forall v \in \{\text{machines outside the subnet}\},\ (v, u) \notin E$$

DHCP client If u is a DHCP client, G satisfies the following.

$$\forall v \in V - \{u\},\ (v, u) \notin E$$

That is since a DHCP client does not have a fixed IP address, no other machines can initiate a connection to it.

The above pieces of information come from the configuration file that will be described in Section 4.6.

4.4.2 Self-stabilizing Spanning Tree Algorithm

The self-stabling spanning tree algorithm [AKY91, AK93] has following features. (1) Each daemon asynchronously builds a spanning tree without knowing the whole network. (2) It can keep a tree even if the network topology changes dynamically.

The algorithm regards the graph as a spanning forest, that is, a set of rooted trees. Initially, this forest consists of a number of single-node trees (each node is a root). Starting from this state, the nodes gradually coalesce into large trees. Eventually, all the nodes in the graph form a single spanning tree. When the network topology changes dynamically, they cope with it by resetting their local states.

Each node maintains three variables: UID, $Parent$, and $Priority$. Subscripts of these variables represent node names. UID_u is a unique identifier of node u and $Parent_u$ a node name of u's parent. $Priority$ is explained shortly.

Each daemon asynchronously connects to its neighbors specified in G, and when two daemons find them to be in different trees, these two trees get merged. Omitting some details, $Priority$ determines how they are merged as follows.

- $Priority_u$ is initialized to UID_u.
- When u finds that its neighbor v has a higher $Priority$ value, u becomes a child of v's tree. Then $Priority_u$ becomes equal to $Priority_v$.

As a result, trees with higher $Priority$ overrun trees with lower ones, and finally the algorithm constructs a single spanning tree with the highest $Priority$.

Figure 4.4 illustrates a process of tree construction. A dashed edge from u to v indicates that $(u, v) \in E$. A solid edge from u to v indicates that v is u's parent. The value of each node shows its $Priority$, which is updated during tree construction.

For example, the node that has $Priority = 1$ initiates a connection to the node that has $Priority = 8$. Then, their $Priority$ become 8. Eventually, all the nodes have $Priority = 9$, and they finish tree construction.

We refer the reader to [AKY91, AK93] for the details of the algorithm.

4.4.3 Routing Algorithm

The shell needs to calculate a path to any participating machine in order to submit a job, redirect input/output of a command, and so on. For this purpose, the shell keeps track of the whole network topology by receiving fragments of topology information (e.g., a list of connections that a daemon currently maintains) from daemons. Every time the network topology changes, the daemons send their new information to the shell and the shell updates its topology information.

Initially, the daemons do not know the home host (where the user logged in). Thus each daemon needs to calculate the path from itself to the home in the spanning tree.

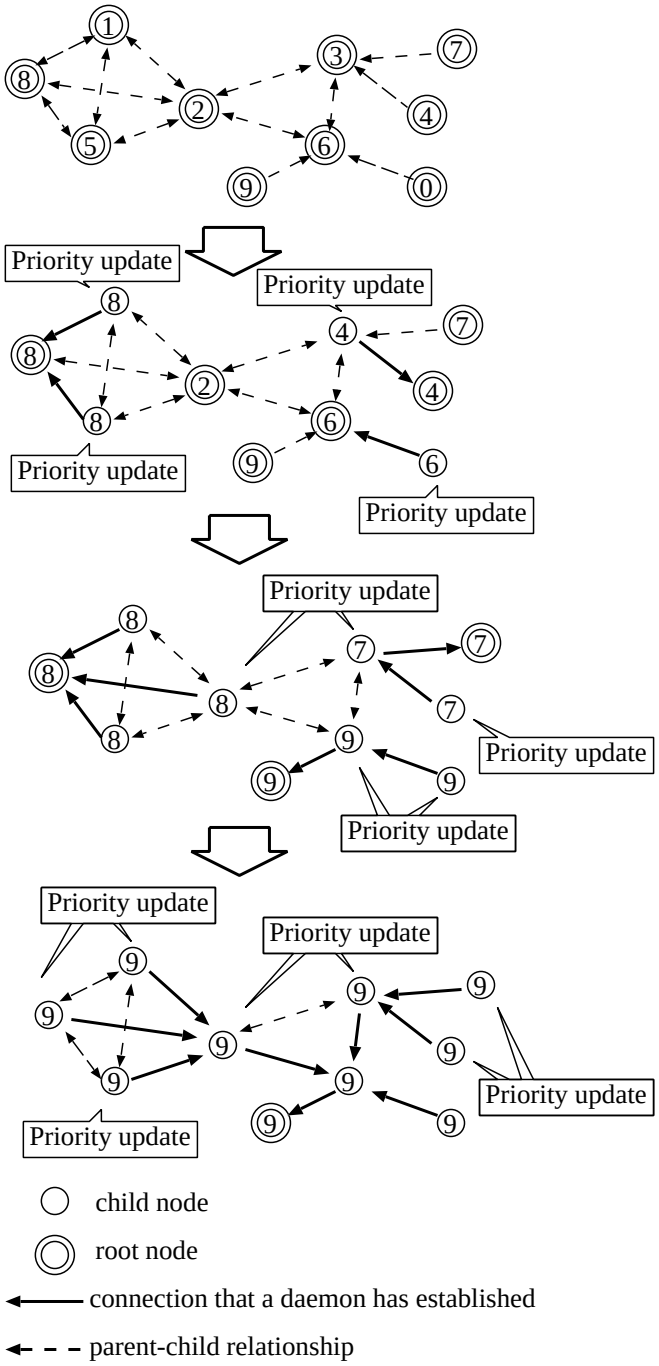

Figure 4.4: Process of spanning tree construction

Each node u maintains an additional variable $ToHome_u$, which will be set to its neighbor one hop closer to the home in the spanning tree. In the sequel, we call such a neighbor its parent. Do not confuse it with the parent used in building the spanning tree. Each node periodically and asynchronously runs the algorithm depicted in Figure 4.5 at regular intervals to determine its parent, which works as follows.

- Initially $ToHome_u$ =nil for all u. It is reset to nil whenever a path to the home host is broken.
- If u is the home host, $ToHome_u$ becomes equal to u ([A] in Figure 4.5).
- If u is a leaf of the tree, its parent is determined uniquely; u regards its neighbor as its parent and sends a message to the home host via the neighbor ([B] in Figure 4.5).
- If all u's neighbors except v are u's children, v must be u's parent (c.f., w is u's child only if $ToHome_w = u$). In such a case, u sends a message to the home host via v ([C] in Figure 4.5).
- If u's neighbor v is not u's child and v satisfies $ToHome_v \neq$ nil, v has already detected a route to the home host and this route does not include u. Thus, u regards v as its parent and sends a message to the home host via v ([D] in Figure 4.5).
- If u does not satisfy all the above conditions, u cannot find a route to the home host yet; $ToHome_u$ is set to nil ([E] in Figure 4.5).

4.5 Implementation Based on Phoenix

We describe the implementation of the inter-process communication with the Phoenix library. The inter-process communication is simply implemented in the following manner:

1. Individual nicknames of hosts are mapped to virtual node names of Phoenix in a one-to-one fashion. The current implementation uses 62-bits prefix of nicknames as virtual node names.
2. When booting, daemons assume virtual nodes corresponding to their nicknames. The daemons construct an overlay network and calculate message forwarding routes as mentioned in Section 2.3.
3. When a user attempts to submit a job to a remote machine, the local daemon receives a request and submits the job to the target remote machine via the overlay network.

if (u is the home host)
then
  $ToHome_u \leftarrow u$ [A]
else if (u is a leaf)
then
  $ToHome_u \leftarrow u$'s parent [B]
else if ($|\{v \in Neigh_{\mathbf{u}} | ToHome_v \neq u\}| = 1$)
then
  $ToHome_u \leftarrow$ the element of the above set [C]
else if ($\exists v \in Neigh_{\mathbf{u}}$,
    ($ToHome_v \neq$ nil) **and** ($ToHome_v \neq u$))
then
  $ToHome_u \leftarrow$ any such v [D]
else
  $ToHome_u \leftarrow$ nil [E]

where:
$Children_{\mathbf{u}} = \{v | Parent_v = u\}$
$Neigh_{\mathbf{u}} = Children_{\mathbf{u}} \cup \{Parent_u\}$

Figure 4.5: Algorithm for node u to detect a route to the home host

4.6 Discussion

Manual Configurations. A user needs to configure VPG manually before using it.

First, a user must boot daemons on all the available machines that s/he would like to use. The user can boot these daemons in any order. They create and keep some bi-directional (TCP) connections to make a single connected graph of all the machines. After a while, they construct a spanning tree and stop establishing new connections. Whenever the topology of the network changes, the daemons re-construct a spanning tree by adding/removing some connections.

Next, the user must run a shell on a host, which s/he initially logged in. We call it the *home host* in the rest of the chapter. The home host keeps track of the whole network topology by receiving fragments of topology information (e.g., a list of connections that a daemon is keeping) from all the daemons. By using this information, the shell can detect a route to any participating machine for job submissions, redirections, and pipes.

Then, the user needs to set up SSH. Many firewalls block connections to all the unprivileged ports, and in this case, SSH is usually the only way to log on machines behind them. Thus VPG daemons use SSH to establish connections to such machines when regular connections are not allowed. Because they need to create SSH connections automatically without entering the user's password, SSH must use the public key authentication with an empty pass-phrase. The user, on the other hand, may want stronger authentication than an empty pass-phrase. Thus we are planning to allow non-empty pass-phrase authentication in the next implementation.

Finally, the user must write a configuration file and give the following information to VPG daemons.

Nickname As previously mentioned, each daemon needs a *nickname*, a name of the machine that does not change over time. Each nickname must be unique throughout all the available machines. Basically, a user must manually give a nickname to each machine. However, if machines have DNS host names and nicknames are not given manually, the same nicknames as their host names are automatically assigned to them.

Port number A daemon tries to contact other daemons at this port.

List of connections Network configuration is specified by a list of connections each daemon can initiate. For example, if a machine has only a private IP address, connections to this machine from outside its subnet will not be listed. Similarly, connections *to* DHCP clients will not be listed, but those *from* them will. Each connection is labeled either as 'regular' or 'ssh', the former indicating it can be a regular connection and the latter it should tunnel through SSH. Daemons construct a spanning tree by selecting connections from this list.

The amount of configuration is fairly large and may be cumbersome for a user. Note, however, that a configuration file is typically written only once, and need not be very precise. For example, it does not affect correctness to add non-existing machines to the list. They are simply regarded as down machines, and the spanning tree algorithm can tolerate them. Similarly, it does not affect correctness to list connections that are actually blocked and not to list connections that are actually possible.

Currently, they are nevertheless required for performance. Listing too many machines or connections that do not exist causes daemons to try many connections that only fail. We are planning to address this issue in our future work. The spanning tree algorithm operates in such a way that each daemon only needs information about allowed connections adjacent to it. Thanks to this distributed nature of the algorithm, probing necessary configuration online should not involve much technical difficulty.

Security Issues. VPG runs at the user level and does not modify existing security policies; it works completely under the existing security policies. For example, a user needs her/his own account on machines in which s/he wants to boot VPG daemons. If SSH is the only method to access remote machines, VPG accesses these machines with it.

Though VPG satisfies the existing polices, it may weaken system security This is because common security policies usually allow a user to access a different user's daemon. A malicious user may access different users' daemons and execute dangerous commands with their privilege. Hence, VPG daemons should authenticate each other when connections between them are established.

Note that VPG creates new connections only when it constructs a tree. VPG need not perform an authentication whenever submitting a job (an authentication is assumed to be valid while connections are maintained). As a result, even if it performs an involved authentication, VPG does not incur large overhead for every job submission. This mechanism is different from common job submission tools (e.g., Globus [CFK+98]). These tools establish a new connection and perform an involved authentication whenever submitting jobs.

Currently, authentication between daemons has not yet been implemented. We are planning to implement this mechanism in the next release.

4.7 Experiments

4.7.1 Experimental Environments

We ran the original implementation of VPG in the network shown in Figure 4.6. The network consists of three subnets, and machines are equipped with several operating systems (Solaris, Linux, and IRIX) and CPU architectures (SPARC, x86, PowerPC, and

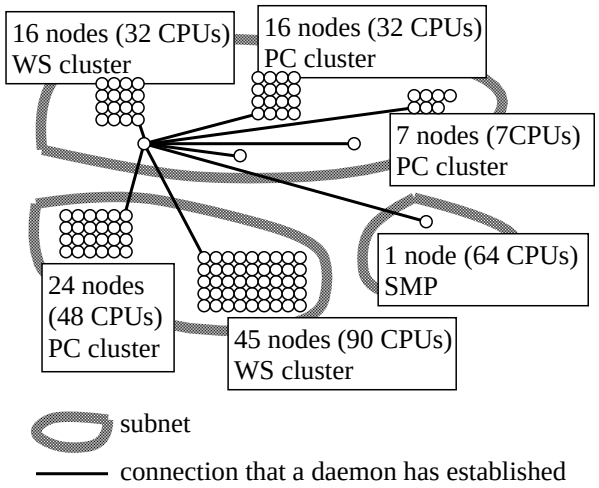

Figure 4.6: Experimental environments

MIPS). VPG daemons on about 100 nodes and demonstrates its feasiblity. In this experiment, daemons constructed a spanning tree that had a diameter of five.

4.7.2 Job Submission Time

We compared VPG with three other job submission tools: rsh, SSH, and globus-job-run (globus-job-run is a remote job submission tool provided by Globus [CFK+98]). Rsh used Rhost authentication, SSH public key authentication (1024-bit RSA), and globus-job-run X.509 authentication (1024-bit RSA).

We measured a turn around time of a small job submission. This time is almost equal to overhead of a remote job submission. In addition, we measured the time of a job submission to machines several hops away from a local host with rsh, SSH, and VPG. Rsh and SSH submit a job to a destination machine by relaying job submissions several times (e.g., `ssh hostA 'ssh hostB command'`).

Figure 4.7 illustrates the result of this experiment. The overhead of VPG was less than that of SSH and globus-job-run. In addition, the overhead of job submissions except VPG increased with the number of relays. This is because the overhead of the job submission was mainly caused by authentication. SSH and globus-job-run create a new connection and perform an involved authentication using a public key whenever submitting a job. In contrast to these tools, VPG does not require authentication whenever submitting a job as mentioned in Section 4.6. Authentication is performed only when it creates a logical network. Jobs and their input/output are routed via connections that have already been created. Hence, the overhead of SSH and globus-job-run were larger than that of rsh and

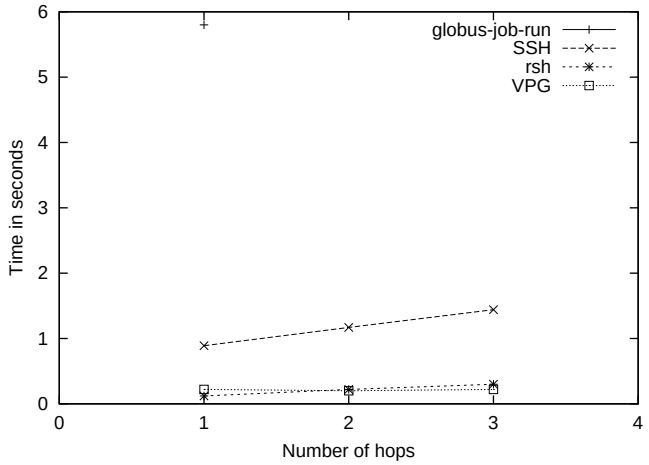

Figure 4.7: Comparison of job submission overhead of rsh, SSH, and globus-job-run

VPG.

4.8 Related Work

Remote Job Submission Systems. Although many remote job submission systems targeting at clusters and computational grids have been developed (e.g., Globus [FK97], Condor [LLM88], Nimrod [BAG00], GLUnix [GPR+98]), none of them are focusing on integrating many 'desktop' resources that are typically configured without DNS names and with DHCP/private IP addresses. Such machines constitute a large fraction of compute resources. The original Globus is blocked by typical firewall configurations and cannot submit jobs from outside firewall to the inside. In addition, because globus-job-run needs to specify a target with its host name or IP address, it is difficult to submit jobs to a machine that has no unique and consistent IP address. Both Condor and Nimrod can automatically submit jobs to remote machines that satisfy the requirements of the jobs. However, their implementation also requires that the machine a user logs in can initiate a direct connection to all the remote machines.

SSH [Opeb] is one of the most common tools for submitting a job to machines behind a firewall. However, when the number of available machines is large and complex administrative restrictions are imposed on these machines, SSH cannot utilize them easily and efficiently. For example, s/he may need to use SSH several times in order to reach a target machine. In addition, it incurs large overhead to initiate SSH connections every time s/her submits a job, as we have described in Section 4.7.

Traversal of NATs and Firewalls. SOCKS [LGL+96] is closest to our work, but has a limited functionality. It is a proxy protocol that provides general framework for circuit level gateway. In typical scenarios, machines inside NAT/firewalls connect to a SOCKS server and machines outside NAT/firewalls reach these machines through the server. There are two main differences between SOCKS and VPG. First, SOCKS does not have nicknames, so naming DHCP clients remains as an issue, and ensuring the uniqueness of local IPs (in different subnets) is up to the user. Second, more importantly, forwarding connections through multiple SOCKS servers is supported but must be configured manually. Therefore it will be difficult to manage hundreds of machines across many (e.g., > 5) subnets. VPG provides a unique naming scheme for clients with DHCP/private addresses. In addition, its communication mechanism minimizes the need for manual configuration.

RMF (Resource Manager beyond Firewall) [TSH+99] is a modified Globus Resource Allocation Manager (GRAM), which can utilize resources behind firewalls. For example, RMF supports job submissions from outside firewall to the inside, whereas the original Globus does not. RMF implements this function by using a proxy that relays messages beyond a firewall. It is similar to SOCKS and thus incurs the same problems as SOCKS does. RMF requires cumbersome manual configuration to manage machines across multiple subnets and has no naming scheme for private IPs and DHCP clients.

Virtual Private Network (VPN) [SWE98] is a mechanism to securely connect multiple subnets through a public network. Because VPN usually requires manually modifying administrative policies (e.g., tunneling), VPG mainly differs from VPN in that VPG automatically constructs a private network at the user level and places major emphasis on a remote job submission.

4.9 Summary

In this chapter, we have described *Virtual Private Grid (VPG)*, a shell that can easily utilize hundreds of machines distributed over multiple subnets. The shell gives each machine a unique nickname that does not depend on a DNS name or a fixed IP address. In addition, it provides a job submission, redirection, and pipe on any nicknamed machine. VPG implements these functions by constructing a self-organizing network among machines and by forwarding messages on the network.

Our future work is to provide easier and more efficient utilization of remote computational resources.

Reduction of the amount of manual configuration As we have described in Section 4.6, VPG requires some manual configuration (e.g., a user needs to write a configuration file). We are planning to reduce the amount of configurations by automatically discovering available machines and possible connections wherever possible.

Authentication between daemons As mentioned in Section 4.6, authentication between daemons should be performed. We are planning to implement simple authentication mechanism using a method similar to MIT-MAGIC-COOKIE-1.

Daemon sharing An original design criteria of VPG is to run daemons at the user level, thereby not requiring help from system administrators. This design, on the other hand, may impose a large overhead on the system if many users run their daemons. We can fix this problem by running a daemon at the root privilege that, with an appropriate authentication, forks a user process on demand. We are going to implement VPG so that it can be run either way. This is not a peculiar problem to VPG, but a common problem in any network service. Whether a service is run with the root privilege or with individual user's privilege is a matter of choice at each host, based on the popularity of the service, system administration policy, and so on.

Automatic resource selection With hundreds of machines, a user wants jobs and data to be distributed over remote machines automatically without explicit annotations. We are planning to develop a simple task placement algorithm that takes the location of input/output files, communication through pipes, and machine architecture into account.

Chapter 5

A Virtual Machine Monitor for Providing a Single System Image

Overview

We designed and implemented a virtual machine monitor that virtualizes a shared-memory multi-processor machine on a commodity cluster. It permits parallel applications to run on them without modifications. In addition, commodity operating systems that support multi-processors (e.g., Linux) can be installed on a virtual machine with only a small amount of modification. We built a virtual eight-way multi-processor machine on eight physical machines, with Linux installed. We present here the results of parallel, coarse-grained tasks that ran on this system. The experimental results demonstrated the feasibility of our approach.

5.1 Introduction

With the recent increase in the performance/price ratio of personal computers (PCs), there is rapidly expanding interest in the use of computing clusters composed of commodity computers. While clusters of commodity PCs can range from small (less than 64 nodes) to large (approaching 10,000 nodes), small clusters are gaining widespread use, particularly at the workgroup and departmental levels.

A major problem for utilizing small commodity clusters has been the complexity of resource management. Without global (cluster-wide) mechanisms for resource allocation, it is difficult to efficiently utilize resources such as processors, memory, and disks.

To overcome this problem, we propose a method for providing a single system image (SSI) on top of a cluster. While various systems (e.g., SCore [SCo], Condor [LLM88]) have been proposed to provide an SSI, we focus on a technique for achieving an SSI with

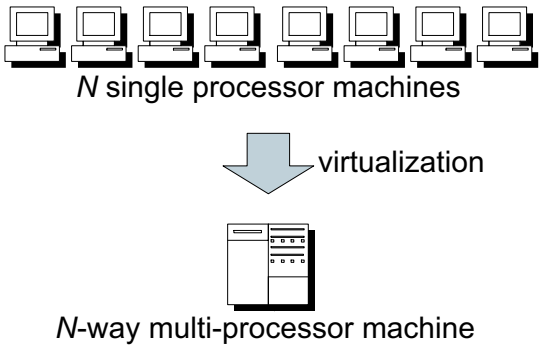

Figure 5.1: Creation of a virtual multi-processor machine

hardware virtualization. More specifically, we designed and implemented a virtual machine monitor (VMM) called *Virtual Multiprocessor*. Like existing VMMs [VMw, WSG02, BDF+03, BDR97, ES03, HBS02, Use, Mic, Coo], Virtual Multiprocessor takes complete control of the machine hardware and creates virtual machines, each of which behaves like a complete physical machine that can run its own operating system. In contrast to existing VMMs, Virtual Multiprocessor has two distinguishing features. First, it virtualizes a shared-memory multi-processor machine on a commodity cluster. For example, it gives a user the illusion of an N-way multi-processor machine on top of a collection of N single-processor machines (See Figure 5.1). Inside the virtual machine, the user installs an operating system that supports multi-processor machines and executes parallel programs on the operating system. Second, Virtual Multiprocessor supports dynamic load balancing. By migrating processors of a virtual machine, our VMM enables the virtual machine to provide a fixed number of processors even if physical machines are added and/or removed dynamically.

Our approach to achieving an SSI has three advantages over existing approaches. First, a wide variety of parallel applications for shared-memory multi-processor systems can run in the virtual machine without any changes to the applications. In particular, a user can write parameter sweep applications or parallel tasks that have directed acyclic graph (DAG) dependency between them using familiar languages and tools designed for shared-memory systems (e.g., parallel `make`, shell script), without resorting to parallel programming languages such as MPI [Mes].

Second, in addition to parallel applications, execution of multiple sequential applications also benefits from our approach. By installing a commodity operating system that supports multi-processors (e.g., Linux) in a virtual machine, the user can manage distributed resources with a familiar interface. If the user forks multiple processes on Linux running inside the virtual machine, these processes are automatically allocated on the virtual machine's processors by the scheduling mechanism of Linux. The processes are then

allocated on the physical machines' processor which the virtual machine's processors are mapped onto.

Third, resource encapsulation with Virtual Multiprocessor provides security and reliability. For example, suppose a cluster is used for server hosting. Since a VMM provides strong isolation between virtual machines and physical machines, the administrator of a VMM can give full control of the virtualized hardware to the users of the virtual machines, without exposing critical resources to danger. This functionality of VMMs greatly facilitates secure and convenient virtual hosting [WSG02, Wal02]. In addition, a snapshot/resume feature of virtual machines reduces the effects of system crashes and break-ins [VMw, WCG04, SPYH03].

Our current implementation of the VMM is designed for the IA-32 architecture. The VMM virtualizes processors, shared memory, and I/O devices as follows. To virtualize processors, the VMM achieves para-virtualization of the IA-32 instruction set architecture (ISA) [BDF+03, ES03]. A guest operating system is statically modified to run optimally on a virtual machine. To virtualize shared-memory, the VMM uses a mechanism similar to software distributed shared memory. The VMM implements the consistency protocol of the shared memory with the virtual memory page protection mechanism of physical machines. To virtualize I/O devices, the VMM prepares a central server that keeps track of the states of all the devices. The VMM communicates with the server whenever a virtual processor issues an I/O operation.

We conducted several experiments to demonstrate the feasibility and performance of our approach. We built a virtual eight-way multi-processor machine with Linux installed on eight physical machines. We ran eight processes that simultaneously calculate a Fibonacci number and measured the execution time. Execution on our virtual eight-way multi-processor machine was about 6.6 times faster than on both virtual and physical one-way processor machines. These results indicate applications that do not require a large amount of the VMM's intervention (e.g., do not access I/O devices very frequently) achieve good performance.

The remainder of this chapter is organized as follows. Section 5.2 presents an overview of Virtual Multiprocessor. Section 5.3 describes the implementation of the virtualization of hardware resources. Section 5.4 gives the details of the memory consistency algorithm. Section 5.5 presents performance measurements. Section 5.6 discusses limitations of our system and solutions for overcoming these limitations. Section 5.7 discusses related work. The final section summarizes the chapter.

5.2 Design

This section presents an overview of Virtual Multiprocessor. First, we describe the basic design decisions. Next, we explain how Virtual Multiprocessor maps virtual resources to

physical resources.

5.2.1 Functionality of Virtual Machines

Functionality of virtual machines built by our VMM is summarized as follows:

- An interface provided by the virtual machines is not at the Application Binary Interface (ABI) level but at the Instruction Set Architecture (ISA) level. The virtual machines provide a complete system environment that supports an operating system along with user processes.
- Both the virtual machines and underlying physical machines are designed for the IA-32 architecture[1].
- The VMM achieves partial virtualization of an underlying machine (i.e., para-virtualization [BDF+03, WSG02]) as opposed to full virtualization [VMw].

Because of para-virtualization, the ISA of the virtual machines is similar, but not identical, to that of underlying hardware. This improves performance, however, the kernel of operating systems running inside the virtual machine requires a small amount of modification. The technique of our VMM for modifying operating systems is similar to that of LilyVM [ES03]. We describe the details of the technique in Section 5.3.

5.2.2 Mapping of Hardware Resources

Virtual Multiprocessor maps hardware resources (processors, memory, and I/O devices) of a virtual machine onto those of physical machines to virtualize a shared-memory multi-processor machine. As shown in Figure 5.2, the resources are mapped in the following manner:

Processors Virtual processors are basically mapped onto physical processors in a one-to-one fashion. N individual processors of a virtual machine are respectively mapped onto a processor of N different physical machines.

Memory A virtual machine's shared memory available to any of the virtual processors is mapped onto a portion of physical machines' memory. Each physical machine needs to reserve M MB of memory to virtualize M MB of the shared memory.

I/O devices I/O devices of a virtual machine are mapped onto devices belonging to one of physical machines. For example, a disk image file located at one of physical

[1] In particular, we chose to target the Pentium 4, Intel Xeon, and P6 family processors, which allow a more relaxed memory ordering model than the earlier Pentium processors (e.g., the Intel 486 processor) [Int03].

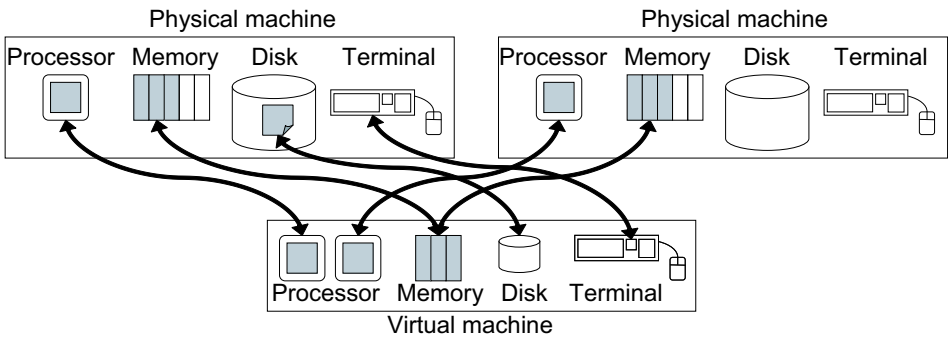

Figure 5.2: Mapping between a virtual machine and physical machines

machines is used as a hard disk image of a virtual machine. A virtual console of a physical machine is used for a serial terminal of a virtual machine.

5.3 Implementation

This section describes how Virtual Multiprocessor virtualizes the IA-32 architecture: processors, shared memory, and I/O devices. The virtualization of shared memory consists of the virtualization of address space and the memory coherence mechanism. First, we explain basic strategy for virtualizing hardware. Next, we explain how individual hardware resources are virtualized. We give just an outline of the virtualization of processors and address space, as it is similar to that of a single-processor virtual machine, particularly LilyVM [ES03]. The virtualization of the memory coherence mechanism and I/O devices is described in more detail because these mechanisms are unique to a multiprocessor virtual machine. Then, we describe a migration mechanism of virtual processors.

5.3.1 Basic Strategy for Virtualizing Hardware

Like LilyVM [ES03] and FAUmachine [HBS02], our VMM is placed on top of a native operating system running on physical hardware and is implemented solely in user mode with no modification to the native operating system. Although this architecture incurs a larger overhead than when the VMM is placed directly on bare hardware [WSG02, Wal02, BDF+03], it overcomes several technical and pragmatic hurdles [SVL01]. First, our system permits virtualization of the Intel Pentium architecture, which is not naturally virtualizable [RI00]. Second, by relying upon a native operating system, it allows a virtual machine to support a diversity of peripheral devices with minimal programming effort.

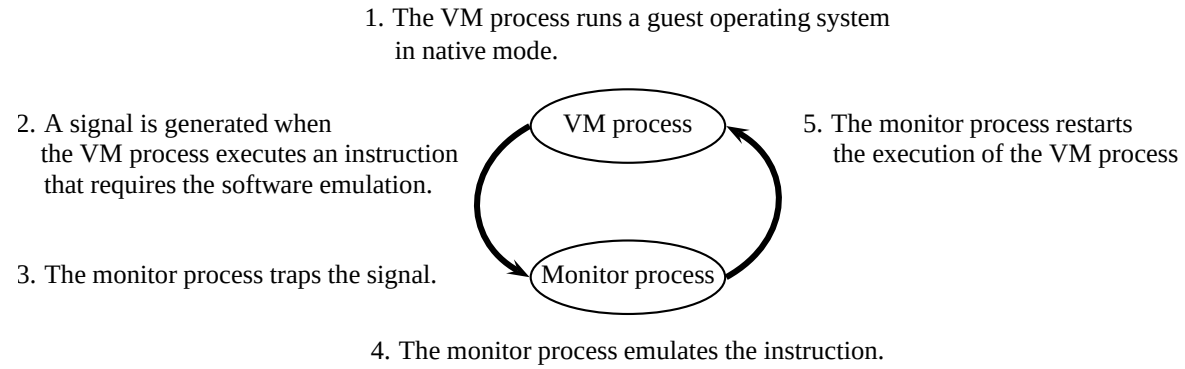

Figure 5.3: A basic execution cycle of the VM process and the monitor process

Third, it allows an operating system installed inside a virtual machine to co-exist with a pre-existing native operating system. Hereafter, we call an operating system running on a virtual machine a *guest* operating system and an operating system running on a physical machine a *host* operating system.

To virtualize hardware resources with no modification to host operating systems, the VMM prepares two user processes for each virtual processor. These user processes are:

VM process The VMM assigns this process to run a guest operating system as one of the processors of a virtual machine. The individual VM processes map assigned virtual processors onto physical processors where the VM processes are running. If the VM process attempts is about to execute an instruction that would interfere with the state of the underlying VMM or a host operating system, a signal is generated by a host operating system. For example, the `SIGSEGV` signal is generated when the VM executes a privileged instruction which cannot be executed by a user process.

Monitor process This process supervises the VM process using the `ptrace` system call. The monitor process intercepts execution of the VM process by trapping a signal generated by the VM process. The monitor process then emulates the instruction executed by the VM process by modifying the state of the VM process's registers and memory.

Figure 5.3 summarizes a basic execution cycle of these processes.

5.3.2 Processor Virtualization

The virtualization of processors consists of (i) the virtualization of instructions that would interfere with the state of an underlying VMM (or a host operating system) and (ii) the virtualization of interrupts and exceptions.

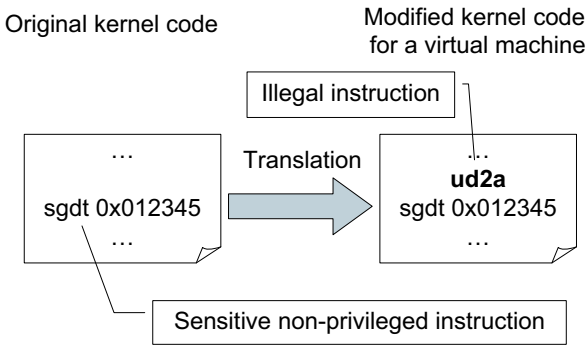

Figure 5.4: Translation of kernel code with a modified assembler

First, we describe the virtualization of instructions that would interfere with underlying systems. As mentioned in Section 5.3.1, a large portion of a virtual processor's instructions is executed by a physical processor without VMM intervention. Only instructions that would interfere with an underlying VMM or host operating system are interpreted by the VMM. These instructions that require VMM intervention are called *sensitive* instructions. For example, instructions that access IA-32 system registers, such as control register 3, are sensitive.

The sensitive instructions are classified into privileged instructions and non-privileged instructions [RI00]. Execution of privileged instructions at the most privileged hardware domain will cause a general protection exception, whereas non-privileged instructions do not cause an exception. For example, the `lgdt` instruction, which loads the value in the source operand into the global descriptor table register (`GDTR`), is a privileged instruction. The `sgdt` instruction, which stores the content of `GDTR` in the destination operand, is non-privileged.

The monitor process traps the execution of privileged instructions and non-privileged instructions in different ways. Trapping of privileged instructions is straightforward: since a VM process runs in user mode, a monitor process needs only to trap exceptions caused by the execution of privileged instructions. On the other hand, trapping of non-privileged instructions is complex and requires modifications to a guest operating system. More specifically, the kernel code of a guest operating system is modified at compile time in such a way that an illegal instruction is inserted before every non-privileged instruction [ES03]. By trapping the exception caused by an illegal instruction, the monitor process intercepts the non-privileged instruction that follows the illegal instruction in the kernel code.

This technique of static modification of kernel code has merits and shortcomings. One of the merits is that numerous operating systems can be hosted with small manual implementation costs. A modified assembler automatically inserts illegal instructions

at kernel compile time. On the other hand, due to static code modification, the VMM cannot support system-level binaries for which the source code is not available. These include binary-only Linux kernel modules and operating systems such as Windows, whose source code is not public.

Second, we describe the virtualization of interrupts and exceptions. To virtualize interrupts and exceptions, the VMM needs to detect and deliver them. The method for detecting interrupts and exceptions varies depending on how they are generated. For example, when a virtual machine generates an exception that can be seen as a signal generated by the VM process, the monitor process detects it by trapping the signal with the `ptrace` system call. When the virtual machine generates an interrupt by accessing its Advanced Programmable Interrupt Controller (APIC), the monitor process detects it by intercepting write access to memory regions which the virtual machine's APIC is mapped onto.

A detected interrupt (or exception) is delivered to an appropriate virtual processor by the monitor process. Basically, the monitor process delivers it to the local VM process. The monitor process makes the VM process enter an interrupt (or exception) handler by looking up the virtual machine's descriptor tables. Only when an inter-processor interrupt is generated, the monitor process delivers it to a specified remote VM process via TCP/IP communication. The delivery of external interrupts triggered by I/O devices is described in detail in Section 5.3.4.

5.3.3 Shared Memory Virtualization

The virtualization of a shared memory requires the virtualization of the address space and a memory coherence mechanism.

First, we briefly explain the virtualization of the address space. A guest operating system running inside a virtual machine expects a zero-based physical address space, as provided by real hardware. To implement such an address space, the VMM needs to virtualize both the segmentation mechanism and the paging mechanism. The former mechanism translates virtual addresses to linear addresses while the latter translates linear addresses to physical addresses. In current implementation, the segmentation mechanism is not fully virtualized. Only the minimal mechanism required to host Linux is implemented. Specifically, reading from and writing to a virtual machine's segment registers are implemented, though translation from virtual addresses to physical addresses is not fully supported. The base address of every segment must be zero inside a virtual machine.

The virtualization of the paging mechanism is implemented in the following manner. First, an individual VM process reserves a portion of its memory for a virtual machine. Then, the VM processes map their pages onto the reserved memory region by looking up the page directory and the page table of the virtual machine.

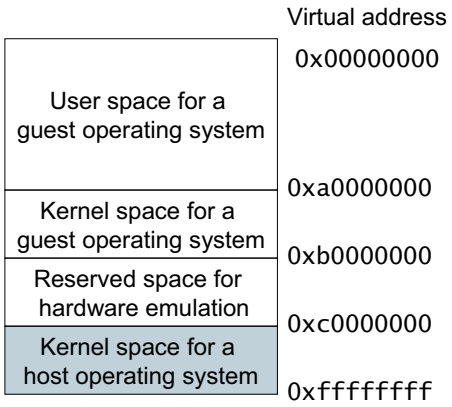

Figure 5.5: Memory layout of the monitor process

More specifically, the VM processes use the `mmap` and `munmap` system calls to update the mapping of pages. Since these system calls incur a large overhead, the system calls are issued only when the modification to the page directory and the page table needs to become valid. For example, suppose that a virtual machine modifies its page table so that page p is mapped onto its physical memory. In this case, the VM process delays issuing the `mmap` system call until the process actually accesses p and the `SIGSEGV` signal is generated. Similarly, the VM process issues the `munmap` system call to release obsolete page mapping only when modification to the page table or page directly becomes valid. For instance, `munmap` is issued to release obsolete mapping when a virtual machine changes the value of the control register 3 or executes the `invlpg` instruction.

The virtual address space that a guest operating system can access is limited. The upper bound of the address space is changed to `0xafffffff` for the following reasons:

- A memory region with lower bound `0xc0000000` and upper bound `0xffffffff` is used for the kernel address space for a host operating system. A VM process, which runs not in supervisor mode but in user mode, is not allowed to access this region by a host operating system.

- A memory region with lower bound `0xb0000000` and upper bound `0xbfffffff` is reserved for hardware emulation. This region is used for storing information required for the virtualization of hardware such as values of system registers.

For the above reasons, a guest operating system is statically modified such that its kernel address space does not overlap with the non-accessible regions. The lower and upper bounds of the kernel address space are changed to `0xa0000000` and `0xb00000000`, respectively (See Figure 5.5).

Next, we describe the virtualization of the memory coherence mechanism. The VMM implements the consistency protocol of the shared memory using the virtual memory page protection mechanism of physical machines. Specifically, the VMM uses the `mprotect` system call to control access to shared pages in such a way that any attempt to perform a restricted access on a shared page generates the `SIGSEGV` signal. Upon trapping this signal, the VMM updates the contents and protection level of the page on the physical machine. The details of the memory sharing mechanism are described in Section 5.4.

Finally, we mention two implementation techniques required for both the virtualization of address space and the virtualization of the memory coherence mechanism. One technique is necessary for the invocation of the `mmap`, `munmap`, and `mprotect` system calls. Since the specification of these system calls allows only a caller process to modify its page mapping and access privilege, for the memory virtualization the system calls need to be issued not by a monitor process but by a VM process. For this reason, a special code for issuing the system calls is mapped on a memory region $[\texttt{0xb0000000}, \texttt{0xc0000000})$ of VM processes [ES03]. When trapping the `SIGSEGV` signal, a monitor process makes corresponding VM process execute the special code.

Another technique is required for the `SIGSEGV` handling. The `SIGSEGV` signals are generated for several reasons, including a page fault exception of a virtual machine and access to a shared page whose privilege is downgraded. Since the way a `SIGSEGV` signals is handled varies depending on the reason the signal is generated, the VMM classifies the signal according to the flow-chart in Figure 5.6.

5.3.4 I/O Device Virtualization

I/O devices currently supported by the VMM include a hard disk and a serial terminal. The supported access methods to these devices include programmed I/O (with `in/out` instructions) and direct memory access (DMA). Access through memory-mapped I/O is not currently implemented.

To emulate I/O devices, the VMM prepares one central server that keeps track of the states of all the devices. We call this the I/O server. The I/O server communicates with monitor processes to emulate the I/O devices. For example, when a guest operating system tries to read a value from an I/O port with the `in` instruction, the I/O server and a monitor process emulate the instruction as follows. First, the monitor process intercepts the execution of the `in` instruction and sends a request to the I/O server. Upon receiving the request, the server reads a value from the specified I/O port and sends it to the monitor process. The monitor process then copies the value to the destination operand of the instruction.

To trigger external interrupts generated by I/O devices, the I/O server checks the state of the devices at regular intervals and tries to find devices that can trigger an interrupt. If such a device is found, the server delivers an external interrupt of the device to a

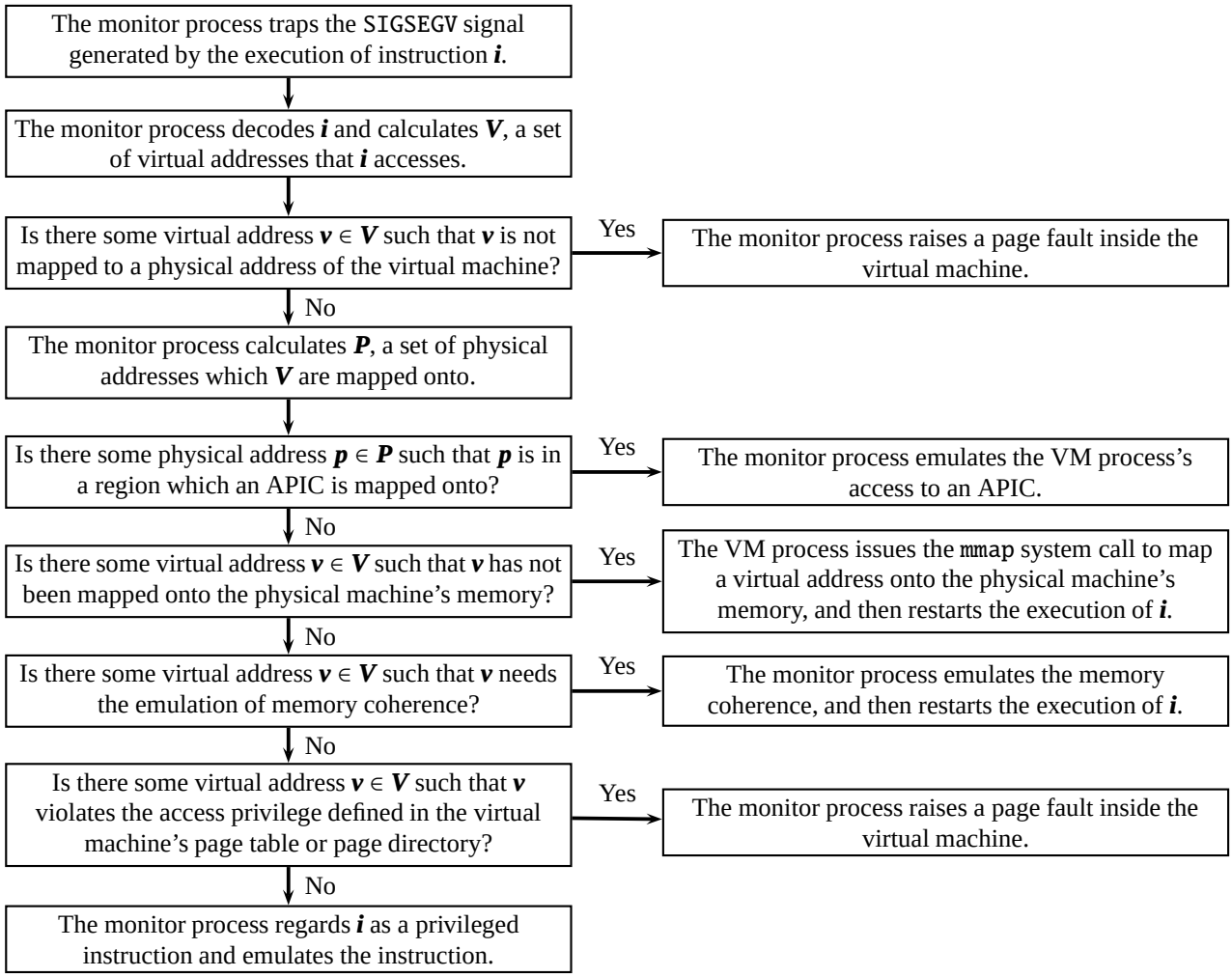

Figure 5.6: Flow chart of the SIGSEGV signal handling

virtual processor in the following manner. First, the server decides a destination virtual processor to which the interrupt is delivered. In the current implementation, the interrupt is delivered to virtual processor v, such that v and the I/O server run on the same host [2]. Second, the I/O server transmits some signal (e.g., `SIGUSR1`) to the VM process corresponding to v to stop its execution. Finally, the monitor process traps the signal and makes the VM process enter an interrupt handler.

5.3.5 Support of Adaptive Parallelism

Virtual Multiprocessor allows a virtual machine to provide a fixed number of processors even if available physical machines are added and/or removed dynamically. For example, an application running inside a virtual machine sees N processors all the time even if the number of available physical machines decreases and becomes less than N.

To provide a fixed number of virtual processors, the VMM maps one or more virtual processors to a physical processor and changes the mapping dynamically. More specifically, the VMM moves virtual processor p from host s to host d in the following manner (See Figure 5.7).

1. Monitor process M_s, which is responsible for p sends requests to other monitor processes not to receive further incoming messages.

2. Monitor processes that receive the request shut down a connection to M_s.

3. If all the connections to the remote monitor processes have been shut down, M_s starts to take a snapshot of p. The snapshot includes states of registers of p, local memory, drivers, and messages that have not yet been handled by M_s.

4. The VMM creates a new VM process and a monitor process M_d at host d.

5. M_s sends the snapshot to M_d, kill its VM process, and exits.

6. M_d establishes connections to other remote monitor processes and resumes the snapshot at host d. M_d also begins to handle messages that were pending.

The current implementation is still naive. For example, migration of virtual processors needs to be triggered manually by a user. Dynamic mapping may cause asymmetric speeds of virtual processors, and may lead to load in-balance. To solve these problems, we plan to implement automatic scheduling of virtual processors and an efficient load balance mechanism of virtual processors using a time ballooning technique [ULSD04].

[2] The scheduling of destinations with dynamic priority has not been implemented yet.

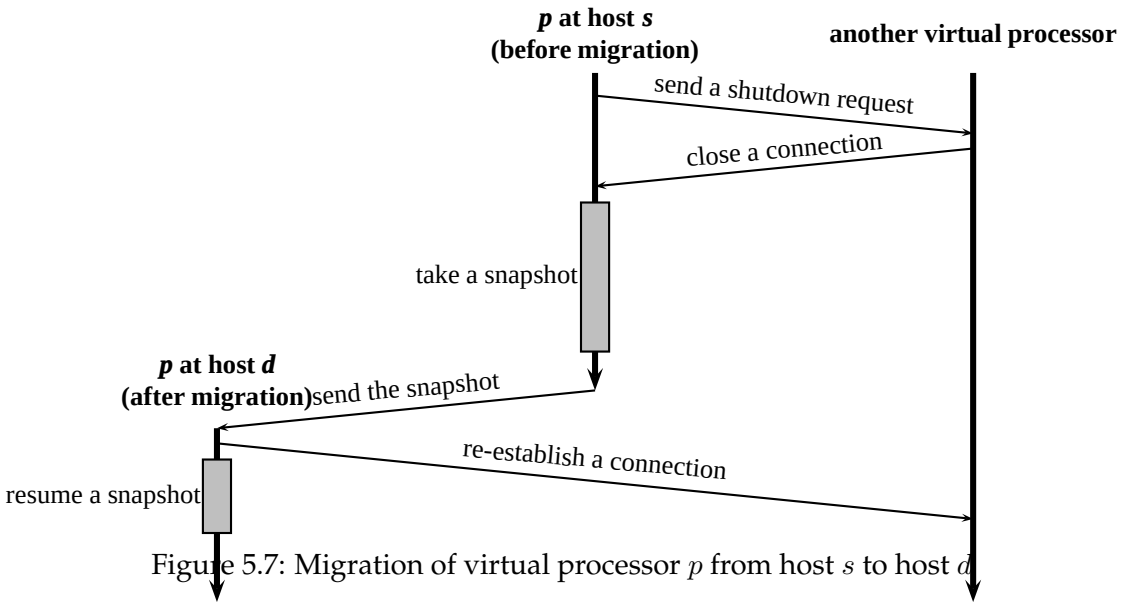

Figure 5.7: Migration of virtual processor p from host s to host d

5.4 Memory Consistency Algorithm

This section explains the virtualization of the memory consistency mechanism in detail. First, we describe the IA-32 memory model. The VMM needs to satisfy this memory model to allow existing programs for the IA-32 architecture to run inside a virtual machine without modification. Then, we present a simple memory consistency algorithm that satisfies the IA-32 memory model. Note that the memory model our algorithm targets differs from that of most existing memory consistency algorithms for distributed shared-memory systems, such as release consistency [KCDZ94, BCZ90].

5.4.1 IA-32 Memory Model

The IA-32 memory model specifies the order in which processors see updates to memory as if they appear to be accessing a single memory. According to the specification of IA-32 [Int03], its memory model is *processor consistency* and guarantees that the following ordering rules apply in multi-processor machines:

- Individual processors use the same ordering rules as in a single-processor machine.
- Writes by a single processor are observed in the same order by all processors.
- Writes from the individual processors are *not* ordered with respect to each other.

Added to the above ordering rules, the IA-32 architecture provides several mechanisms for strengthening or weakening the memory ordering model to handle special

$proc_i$	:	virtual processor i
M_i	:	message queue of virtual processor i
$pages_i^n$	:	nth page of virtual processor i
$p.state$	:	state of page p (`invalid`, `read_only`, or `read_write`)
$p.content$	:	content of page p
$p.owner$	:	processor that own the latest content of page p
$p.copyset$	:	a collection of processors that have a replica of page p
$p.busy$	:	flag which is `true` while page p is being updated

Figure 5.8: Variables for algorithm description

programming situations. These mechanisms include the I/O instructions, locking instructions, the `LOCK` prefix, and serializing instructions that force stronger ordering on processors. For instance, `mfence` is one the serializing instructions. It ensures that every load-from-memory and store-to-memory instruction that precedes the `mfence` instruction in machine code is globally visible when the `mfence` instruction is issued.

5.4.2 Algorithm Description

We explain a simple memory consistency algorithm that satisfies the IA-32 memory model. This algorithm is based on a simple sequentially consistent, multiple-reader/single-write protocol used in Ivy [LH89]. The machine pages of the virtual machine are distributed over nodes such that each node manages a subset of the pages. Figure 5.8 and Figure 5.9 describe the algorithm in more detail. Figure 5.8 shows variables used in the algorithm description. Figure 5.9 describes actions of virtual processor i. When one of the conditions listed on the left side of the figure holds, the corresponding action on the right side of the figure is taken.

We plan to optimize the algorithm by relaxing memory consistency as far as possible why satisfying the IA-32 memory model. This optimization plan is discussed in Section 5.6.

5.5 Experiments

We implemented a prototype of Virtual Multiprocessor and conducted several experiments to demonstrate the performance of our system. This prototype system builds a virtual eight-way multi-processor machine on top of eight physical machines. The virtual machine can host the Linux kernel for SMP and allows various applications (e.g., `gcc`, `make`) to run on Linux.

Guard		Action
$access(i, a, n)$ $\wedge$ $violation(pages_i^n, a)$	$\Longrightarrow$	stop the execution of the VM process; send $\langle$**fetch**$, n, a, i\rangle$ to $manager(n)$;
$\langle$**fetch**$, n, a, s\rangle \in M_i$ $\wedge$ $pages_i^n.busy =$ false	$\Longrightarrow$	remove $\langle$**fetch**$, n, a, s\rangle$ from M_i; let p be $pages_i^n$; $p.busy :=$ true; **match** a **with** read $\Rightarrow$ send $\langle$**invalidate**$, n, a, s, p.owner\rangle$ to $p.owner$; write $\Rightarrow$ **forall** $x \in p.copyset$ such that $x \neq proc_s \vee x = p.owner$ **do** send $\langle$**invalidate**$, n, a, s, p.owner\rangle$ to x; **end**

$\langle$**invalidate**$, n, a, s, o\rangle \in M_i$	$\Longrightarrow$	remove $\langle$**invalidate**$, n, a, s, o\rangle$ from M_i;
		let p be $pages_i^n$;
		match a **with**
		read $\Rightarrow p.state :=$ read_only;
		write $\Rightarrow p.state :=$ invalid;
		end;
		if $o = proc_i$ **then** send $\langle$**ack**$, n, a, p.content\rangle$ to $proc_s$;

$\langle$**ack**$, n, a, c\rangle \in M_i$	$\Longrightarrow$	remove $\langle$**ack**$, n, a, c\rangle$ from M_i;
		let p be $pages_i^n$;
		$p.content := c$;
		match a **with**
		read $\Rightarrow p.state :=$ read_only;
		write $\Rightarrow p.state :=$ read_write;
		end;
		send $\langle$**finish**$, n, a, i\rangle$ to $manager(n)$;
		restart the execution of the VM process;

$\langle$**finish**$, n, a, s\rangle \in M_i$	$\Longrightarrow$	remove $\langle$**finish**$, n, a, s\rangle$ from M_i; let p be $pages_i^n$; **match** a **with** read $\Rightarrow p.copyset := p.copyset \cup \{ proc_s \}$; write $\Rightarrow p.copyset := \{ proc_s \}$; $p.owner := proc_s$; **end**; $p.busy :=$ false;

where

$manager(n)$	:	manager of the nth page(e.g., $manager(n) = proc_{n \bmod N}$ where the number of processors is N)
$access(i, a, n)$	:	This predicate holds when processor i accesses with a (read or write) to nth page
$violation(p, a)$	$\equiv$	$p.state =$ **invalid** $\vee$ ($p.state =$ **read_only** $\wedge$ $a =$ **write**)

Figure 5.9: Simple memory consistency algorithm (for virtual processor i)

Table 5.1: Sequential benchmark programs and their execution time on a physical and a virtual single-processor machine (units: seconds)

Name	Description	Execution time (physical)	Execution time (virtual)	Overhead ratio
`fib`	Calculate a Fibonacci number	22.6	22.1	0.97
`getpid`	Issue `getpid` $100,000$ times	0.05	18.1	354
`ls`	List file information	0.03	6.64	255
`gcc`	Compile a C program	0.14	0.98	6.81

Specifically, we conducted the following experiments. First, we ran several sequential programs on a virtual single-processor machine to measure the overhead of hardware virtualization, excluding the memory coherence mechanism. The hardware virtualization involves emulation of sensitive instructions, access to I/O devices, and so on. Second, we ran parallel, coarse-grained tasks on a virtual multi-processor machine to measure the overhead of the memory coherence mechanism. Third, we ran NAS parallel benchmarks on a virtual dual-processor machine to measure performance of parallel programs that are more complex than parallel, coarse-grained tasks. Finally, we measured overheads of migration of virtual processors.

All the experiments were conducted on 2.4 GHz Intel Xeon machines with 2 GB RAM and a 1 Gigabit Ethernet NIC. In Section 5.5.1 and Section 5.5.2, Linux 2.4 was used for both the host operating system and the guest operating system. In Section 5.5.3 and Section 5.5.4, Linux 2.6.8 was used for both the host operating system and the guest operating system.

5.5.1 Execution of Sequential Programs

We ran several sequential programs on a virtual single-processor machine to measure the overhead of hardware virtualization, excluding the memory coherence mechanism.

Table 5.1 shows a description of benchmark programs and their execution times on a physical machine and a virtual machine. Although the experimental results indicate that overheads incurred by the execution of `getpid`, `ls`, and `gcc` are large, the overheads can be reduced, as indicated by the performance of existing IA-32 VMMs (e.g., VMware [VMw], Xen [BDF+03]). Section 5.6 discusses several techniques for reducing the overheads.

5.5.2 Execution of Parallel Coarse-grained Tasks

We measured the execution time of parallel coarse-grained tasks on a virtual multi-processor machine to evaluate the overhead of the memory coherence mechanism. Specifically, we

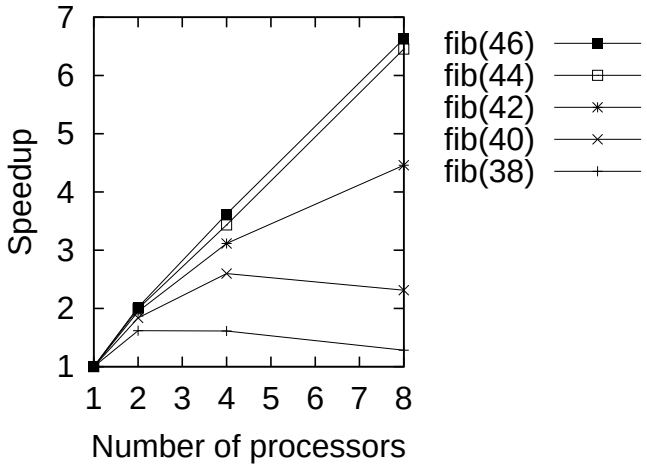

Figure 5.10: Speedup of parallel Fibonacci

ran eight processes that calculate a Fibonacci number simultaneously on a one-way, ..., eight-way virtual multi-processor machine built on top of one, ..., eight physical machines respectively. The overheads of these programs are mainly due to emulation of the following hardware mechanism:

- System calls such as `fork`, used for creating processes.
- Access to a hard disk for loading the executable file and shared libraries.
- The memory coherence mechanism (especially necessary for processes running in kernel mode).

Figure 5.10 shows a speedup of this program. `fib(n)` denotes the calculation of the `n`th Fibonacci number. As shown in this figure, the program achieved a better speedup as tasks were coarser. The execution of `fib(46)` on an eight-way multi-processor machine is about 6.6 times faster than on a one-way processor machine.

Table 5.2 gives the breakdown of the execution of Fibonacci numbers. 'Total' denotes the total execution time. 'Native' denotes how long the virtual machines ran in native mode. 'Shmem' denotes the time spent for the virtualization of the memory coherence mechanism. 'Misc' denotes the time spent for the hardware virtualization, excluding the memory coherence mechanism. 'Idle' denotes how long the virtual machines executed the `hlt` instruction. For more than one processor, the table shows the average of the execution times of individual processors. Table 5.2 indicates that the overhead of `fib(40)` and `fib(44)` is mainly caused by the virtualization of the memory coherence mechanism, which becomes larger as the number of processes increases.

Table 5.2: Breakdown of execution time of `fib(40)` and `fib(44)` (units: seconds)

`fib(40)`

# of procs.	Total	Native	Shmem	Misc	Idle
1	26.2	25.8	0.0	0.4	0.0
2	14.3	12.9	0.7	0.4	0.2
4	10.1	6.5	2.1	0.2	1.3
8	11.3	3.6	3.6	0.1	3.9

`fib(44)`

# of procs.	Total	Native	Shmem	Misc	Idle
1	180.0	177.8	0.0	2.2	0.0
2	90.3	87.9	1.0	1.1	0.3
4	52.4	43.7	3.0	0.4	5.3
8	27.9	22.1	3.7	0.1	2.0

Total : total execution time
Native : time for a virtual machine to run native mode
Shmem : time for the virtualization of the memory coherence mechanism
Misc : time for the virtualization, excluding the memory coherence mechanism
Idle : time for a virtual machine to execute the `hlt` instruction

We further investigated the overhead incurred by the virtualization of the memory coherence mechanism for `fib(44)`. We measured the distribution of virtual addresses fetched by the monitor processes for memory sharing (Figure 5.11) and the distribution of times to complete individual page fetch requests (Figure 5.12). Figure 5.11 indicates that page fetch requests frequently occurred at the beginning and end of `fib(44)` (in both user and kernel mode). Note that the base address of the kernel space of the guest operating system is changed to `0xa0000000`, as mentioned in Section 5.3.3. Figure 5.12 shows that some fetch requests took tens of milliseconds to complete, whereas most page fetches were completed in less than ten milliseconds. These experimental results indicate that the overhead of the parallel coarse-grained tasks is due to frequent page fetches caused by false sharing.

5.5.3 Execution of NAS Parallel Benchmarks

We built a virtual dual-processor machine on two physical machines and ran NAS Parallel Benchmarks (NPB) 2.3 [NASa] on the virtual machine. All the benchmarks are written in OpenMP [NASb]. More specifically, we measured execution time of Embarrassingly Parallel (EP), Conjugate Gradient (CG), LU Decomposition Simulated CFD Application

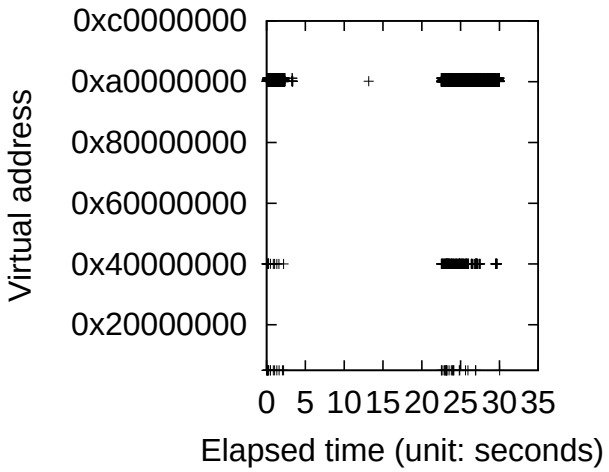

Figure 5.11: Distribution of virtual addresses fetched by the monitor processes for memory sharing (for `fib(44)`)

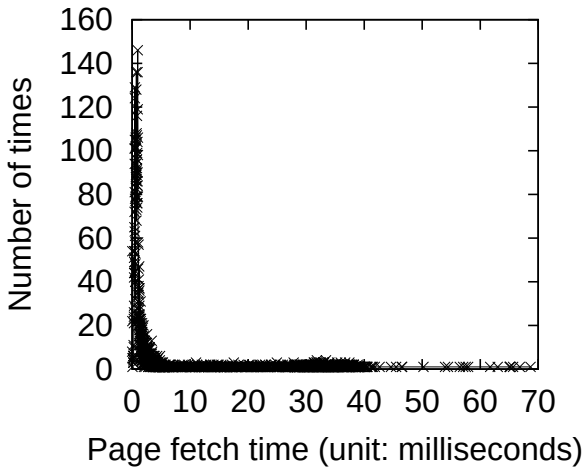

Figure 5.12: Distribution of elapsed times to complete individual page fetch requests (for `fib(44)`)

Table 5.3: Execution time of the NAS Parallel Benchmarks 2.3 written in OpenMP (units: seconds)

Benchmark	Class	Physical machine (single processor)	Virtual Machine (single processor)	Virtual Machine (dual processor)
EP	S	6.35	6.73	7.36
	A	101.48	102.69	58.74
CG	S	0.24	1.11	48.76
	A	5.23	14.00	N/A
LU	S	0.16	0.64	56.23
	A	332.06	345.52	N/A
SP	S	0.24	0.87	778.68
	A	374.81	392.55	N/A

(LU), and Scalar Pentadiagonal Simulated CFD application (SP) benchmarks.

Table 5.3 shows execution time of the benchmarks on a physical single-processor machine, a virtual single-processor machine, and virtual dual-processor machine. First, we compare execution of the benchmarks on the virtual single-processor machine and on the physical machine. Although the former took a longer time than that on the latter because of interventions the VMM (e.g., context switch, system calls), the overhead became smaller as the problem size became larger. Next, we compare execution on the virtual dual-processor machine and on the physical machine. Though EP of which class size is A achieved an approximately linear speedup, the other benchmarks (CG, LU, and SP) suffered from virtualization overheads. The overheads are mainly caused by communication time required for shared memory emulation.

5.5.4 Migration of Virtual Processors

We evaluated overheads that migration of virtual processors incurs. Specifically, we created a virtual dual multi-processor machine with 32 MB RAM over host X and Y, and moves one of the virtual processors running at X to X'. The monitor process at X took a snapshot of the virtual processor and store it as a local file of an NFS file system. A newly created monitor process at X' got the snapshot via the NFS. We measured a time for the monitor process at X to shut down a communication channel among remote monitor processes and to take a snapshot. We also measured a time for monitor process at X' to resume the snapshot and a downtime of the monitor process at Y (e.g., time to wait for the recovery of connections).

Table 5.4 shows elapsed times of each action that the monitor process took. The ex-

Table 5.4: Breakdown of overheads of virtual-processor migration

Action of the monitor process	Elapsed time in seconds
Shut down connections at X	2.96×10^{-3}
Take a snapshot at X	2.54
Resume a snapshot at X'	7.98×10^{-1}
Wait for connection recovery at Y	2.79

perimental result obviously indicates that taking a snapshot [3] was a dominating cost in migration of virtual processor. To reduce this migration cost, optimization techniques proposed so far (e.g., VMotion [NLH05], Xen [CFH+05]) can be used.

5.6 Discussion

In this Section, we discuss limitations of the current implementation of Virtual Multiprocessor and propose several solutions for overcoming these limitations.

Optimization of Hardware Virtualization. Currently, our VMM is placed on top of a host operating system and is implemented solely in user mode, without any modifications to the host operating system. Although this architecture requires only a small amount of implementation effort, the virtualization of IA-32 architecture (e.g., issues of system calls, access to I/O devices) incurs a larger overhead as shown in Section 5.5.1.

To reduce these overheads, we plan to apply existing optimization techniques developed by numerous IA-32 VMMs (e.g., Xen [BDF+03], CoVirt [KDC03]) to our VMM. For example, the number of context switches caused by the `ptrace` system call can be reduced by placing the VMM directly on bare hardware like Xen. We also plan to port the VMM to other architectures whose designs are more suitable for hardware virtualization [Adv05, Int05].

Optimization of Memory Consistency Algorithm. Since the memory consistency algorithm described in Section 5.4.2 is simple sequentially consistent, we plan to develop an algorithm that relaxes memory consistency as long as the IA-32 memory model can be satisfied.

An example of optimization techniques we are planning is to allow multiple nodes to write to the same page simultaneously. The sequentially consistent algorithm does not allow multiple nodes to write to the same page at the same time since write updates need to become globally visible immediately. In contrast, our optimized algorithm delays write

[3] The size of the snapshot was 41 MB.

updates until a synchronous instruction or an atomic instruction is issued. Note that this relaxation of memory ordering does not violate the IA-32 memory model according to the specification [Int03].

Fault Tolerance. A machine crash is a frequent event in commodity clusters because of the large number of machines involved. Hence, we require that the system can continue to run even if some machines fail. We plan to implement a fault-tolerance mechanism using techniques such as the checkpointing/recovery [EAWJ02] and replication for VMMs [BS96, DH05].

Efficient scheduling for heterogeneous environments. Because the scheduling mechanism of commodity operating systems assumes that underlying processors have the same performance, scheduling is not efficient for heterogeneous environments, which commodity clusters are usual. To alleviate this problem, we plan to develop an efficient scheduling mechanism using a technique such as time ballooning [ULSD04].

5.7 Related Work

Virtual Machine Monitors for Multi-Processor Machines. Recently several VMMs that build a virtual multi-processor machine have been developed. These VMMs include vNUMA [CH05], Virtual Iron [Vira], Disco [BDR97], and VMware ESX Server [Wal02].

vNUMA virtualizes a cc-NUMA machine on top of physical machines with the Itanium architecture. Whereas the memory coherence mechanism of vNUMA based onv Ivy invalidates memory pages synchronously, pages are invalidated asynchronously in Virtual Multiprocessor. This asynchronous page update reduces downtime when a guest operating system is not running though the total number of messages required for each page fetch increases (See Table 5.5).

Virtual Iron [Vira] builds a virtual multi-processor machine on top of clusters. The basic mechanism of Virtual Iron is similar to that of our system. A comparison between Virtual Iron and our system has not been made yet, since details of Virtual Iron are not public.

Disco and VMware ESX Server require a physical machine that has an equal or greater number of processors than they are attempting to virtualize. In contrast, our VMM can build a virtual multi-processor machine regardless of the number of physical processors or the number of machines on which these processors reside. This functionality of our VMM allows users to harness distributed resources efficiently and transparently.

Middlewares and Operating Systems for Providing an SSI. Middleware systems for clusters (e.g., SCore [SCo] and Condor [LLM88]) provide a single software image for

Table 5.5: Comparison of memory consistency algorithm between Ivy and Virtual Multiprocessor. "normal" denotes a process is neither a requester nor an owner.

	access	manager	# of messages	critical path length
Ivy	read	requester	2	2
		owner	2	2
		normal	3	3
	write	requester	$2 + 2c$	4
		owner	$2 + 2c$	4
		normal	$3 + 2c$	5
Virtual Multiprocessor	read	requester	2	2
		owner	3	2
		normal	4	3
	write	requester	$2 + c$	2
		owner	$3 + c$	3
		normal	$4 + c$	4

c: the number of processors that have a page replica except the requester of page fetch

high-performance parallel programming environments. However, the interface provided by these systems differs from that of the commodity operating system. In contrast, our system's interface is same as that of commodity operating system. This functionality greatly simplifies the utilization of distributed resources.

There are several systems (e.g., MOSIX [BL98], OpenSSI [Opec] and Kerrighed [MLV+03]) that enhance the Linux kernel with cluster computing capabilities. Drawbacks that these systems suffer include large implementation costs for kernel modification and difficulty in supporting diverse operating systems.

Software Distributed Shared Memory Systems. Shasta [SGT96] and cJVM [AFT99] are software distributed shared memory systems that transparently support a shared address space across a cluster of workstations. Shasta implements the shared address space by transparently rewriting the application executable to intercept loads and stores. cJVM implements the shared memory space by modifying Java Virtual Machine.

While Shasta and cJVM support only user programs, our system allows an entire operating system for SMP to run on clusters. Furthermore, our system is targeted at the IA-32 architecture, whereas Shasta is targeted at the MIPS architecture and cJVM at Java Virtual Machine.

Simulators and Emulators. SimOS [RHWG95] performs a complete machine simulation that helps investigators better understand the behavior of machines running commercial OSes, as well as any applications designed for these Oses. SimOS supports the simulation of a multi-processor machine in which individual processors are emulated by different Unix processes. Compared with our system, SimOS requires that the nodes where the Unix processes run physically share the same memory.

Bochs [Boc] is an open source IA-32 emulator. Although Bochs also supports the emulation of a multi-processor machine, its emulation is not parallel but sequential, like co-scheduling. As a result, it is not useful for enhancing parallel computing.

5.8 Summary

We have presented Virtual Multiprocessor, a software layer that virtualizes a multi-processor machine on a commodity cluster. The experimental results show that our system achieved good performance for embarrassingly parallel coarse-grained tasks. Since this kind of parallel programs include various useful applications such as parameter sweep applications and parallel `make`, our system can be applied for the wide deployment of commodity clusters.

As mentioned in Section 5.6, we plan a number of extensions and improvements to our system. Furthermore, we plan to evaluate our system using real-world applications such as the SPLASH-2 suite [Sta] and Apache [Apa].

Chapter 6

Conclusions

6.1 Summary of Results

We have presented the middleware systems that enable users to efficiently adapt dynamic changes in their computing environments: (i) Phoenix — a Grid-enabled message passing library for accommodating dynamic addition and removal of machines, (ii) Virtual Private Grid — a remote job-submission system for accessing hundreds of machines seamlessly, (iii) Virtual Multiprocessor — a virtual machine monitor that provides a single system image on cluster systems. Several techniques for improving the scalability of the routing mechanism of Phoenix have been also described.

Our thesis was that these middleware systems have advantages over existing middleware systems from the view point of programmability, scalability, and usability. More specifically, the Phoenix library enables better programmability than existing message-passing systems (e.g., MPI) by providing virtual node names that a programmer maps to processes dynamically. In addition, the implementation of library is comparable in scalability to previous implementations of the message-passing systems. Virtual Multiprocessor achieves good usability by creating a shared-memory multiprocessor machine on a cluster of computers. It allows a user to run parallel applications (and operating systems) for shared-memory multi-processor machines on commodity clusters with no or little modification to their code.

The evaluation of the middleware systems has been also described. The performance of the Phoenix library was evaluated using several benchmark programs, including a parallel ray-tracing program based on Pov-Ray and Integer Sort in the NAS Parallel Benchmark suite. The parallel ray-tracing with a divide-and-conquer algorithm achieved a good speedup with a large number of nodes across multiple LANs (about 78 times speedup using 104 CPUs across three LANs). Experimental results indicated applications with a small task migration cost can quickly take advantage of dynamically join-

ing/leaving resources. In addition, we measured the performance of the routing algorithm of the Phoenix library using 400 nodes in three LANs. The experimental results showed the elapsed time of routing table construction by our algorithm is only about twice as long as that of off-line route calculation. We also demonstrated the feasibility of Virtual Multiprocessor. We built a virtual eight-way multi-processor machine on eight physical machines, with Linux installed, and ran parallel, coarse-grained tasks on the virtual machine. Experimental results indicated applications that do not require a large amount of the VMM interventions (e.g., do not access I/O devices very frequently) achieves good performance.

6.2 Directions for Future Work

Among the possibilities of future work discussed in the previous chapters, the most interesting is to make our middleware systems fault tolerant. Fault tolerance is vitally important for clusters and grids suffering from their high machine-failure rates. Although many failure-recovery algorithms (e.g., replication, checkpointing) have been proposed so far, these algorithms are not straightforwardly applied to our middleware systems. Overhead costs incurred by the algorithms (e.g., long system downtime) need to be reduced.

Another direction is to realize a fusion of Phoenix and Virtual Multiprocessor. We plan to build virtual execution environments that facilitate the deployment of clusters and computational grids using these two middleware systems. More specifically, we plan to build customizable, safe execution environments by combining overlay networks of Phoenix and the resource encapsulation mechanism of Virtual Multiprocessor. The overlay-networks of Phoenix separate virtual networks viewed by users from physical IP networks in which communication among nodes may be restricted by administrative configurations such as firewalls or NATs. The resource encapsulation mechanism allows a user to access consistent, customized application environments that are decoupled from physical resources. These customizable platforms are useful because the heterogeneity of underlying hardware (and software) configurations makes it difficult to run popular programs depending on specific operating systems or libraries. The resource encapsulation also ensures that untrusted users or applications can only compromise their own operating system within a virtual machine, not physical resources. Compared with existing systems that provide virtual machines designed for efficient use of clusters or computational grid (e.g., Xenoserver [FHH+03], SODA [JX03a], Violin [JX03b], PlanetLab [BBC+04], Virtuoso [SD04], vMatrix [AR04], VMPlants [KGZ+04], Virtuozzo [Virb]), our system would be preferable because of its multi-processor virtualization facility and routing facility.

We also plan to improve various aspects of each of our middleware systems. In particular, we plan to improve the scalability of Phoenix to utilize emerging computational

grids approaching thousands of nodes (e.g., Grid'5000 project [CDD+05]). Since experimental results we conducted indicated that scalability limitations of Phoenix are mainly due to overheads incurred by the current implementation of routing table construction, we would like to develop a more efficient routing algorithm. Possible optimization techniques include compaction of routing table, reduction of the number of routing update messages, and efficient connection management that considers underlying physical networks. Although much work on optimization techniques of routing algorithms has been carried out both in theoretical and practical research fields (e.g., compact routing [TZ01]), most of these algorithms have not yet been applied to computational grids. Further studies must to be conducted to reason that the algorithms can be applied to Phoenix.

Appendix A

Specification of the Phoenix Library

This chapter describes APIs of Phoenix library, which is currently implemented for C language. This chapter is organized as follows. Section A.1 explains data types, constants, and global variables. Section A.2 explains initialization and finalization functions. Section A.3 explains virtual node mapping functions. Section A.4 explains message transmission functions. Section A.6 explains low-level interfaces that allows flexible initialization and finalization of the library. Section A.7 explains machine/network configurations that are required to use the library.

A.1 Data Types, Constants, and Global Variables

A.1.1 Data Types

`ph_vp_t` `ph_vp_t` denotes a single virtual node. This data type is mainly used for specifying a destination address of messages. For example, the `ph_send` function uses `ph_vp_t` as the type of its first argument, which specifies a message destination.

`ph_vp_t` is implemented as a 64-bit integral type. In the 32-bit Linux, it is defined as `long long`.

`ph_vps_t` `ph_vps_t` denotes a collection of virtual nodes. This data type is mainly used for mapping virtual nodes to processes. For example, `ph_assume_vps` and `ph_release_vps` use `ph_vps_t` as the type of their first argument, which specifies a set of virtual nodes to be assumed and released respectively.

A virtual node set is created by functions such as `ph_vps_create_singleton` and `ph_vps_create_range`. A newly created set of virtual nodes is allocated in heap space by the Phoenix library. After the finish of its use, the destruction function `ph_vps_destroy` should be invoked for recycle of the heap space.

field name	type	description
body	void *	pointer to the body of the message
len	size_t	length in bytes of the body
dest	ph_vp_t	message destination
tag	int	message tag

Table A.1: Fields of ph_msg_t

ph_vps_iter_t ph_vps_iter_t is a type for iterator over a collection of virtual nodes. This data type is mainly used for iteratively accessing a virtual nodes set of which type is ph_vps_t.

ph_msg_t ph_msg_t is a type for messages that are sent and received in the context of Phoenix. This data type is mainly used as the type of the return value of the ph_recv function. ph_msg_t is designed as a *pointer* to a data structure of which fields are listed in Table A.1.

A newly created message is allocated in heap space by the library. After the finish of its use, the destruction function ph_vps_destroy should be invoked for recycle of the heap space.

A.1.2 Constants

PH_INVALID_VP PH_INVALID_VP (= -1LL) is a constant value that denotes an invalid virtual node. For example, the ph_vps_get_min_elem and ph_vps_get_max_elem functions return PH_INVALID_VP if an empty set of virtual nodes is given and they cannot return any valid virtual node.

PH_MSG_ANY_TAG PH_MSG_ANY_TAG (= 0) indicates a wild card of a message tag. A message sent with PH_MSG_ANY_TAG will be received with any tag. A receive operation with PH_MSG_ANY_TAG receives a message sent with any tag. Section A.4 details how message tags work.

A.1.3 Global Variables

PH_VP_LOWER_BOUND PH_VP_LOWER_BOUND is the lower bound of a virtual node space that an application is allowed to manipulate. This value is specified by the ph_init function.

Note: PH_VP_LOWER_BOUND = ph_get_vp_lower_bound() holds. □

PH_VP_UPPER_BOUND PH_VP_UPPER_BOUND is the upper bound of a virtual node space that an application is allowed to manipulate. This value is specified by the ph_init function.

Note: Note that PH_VP_UPPER_BOUND = ph_get_vp_upper_bound() holds. □

A.2 Initialization and Finalization Functions

This section describes the ph_init and ph_finalize functions, which are used for the initialization and finalization of the Phoenix library respectively. The ph_init (or ph_finalize) function is basically called before (or after) the invocation of any other Phoenix APIs.

void ph_init (*ph_vp_t* ***lower***, *ph_vp_t* ***upper*****, *const char* ****config_file*****, *const char* ****config_tag*****, *const char* ****session*****, *const char* ****msg_log_file*****)**

ph_init(lower, upper, config_file, config_tag, session, msg_log_file) initializes the Phoenix library. The initialization of the library takes three steps: (1) invocation of the Phoenix runtime system, (2) set up of contact points (with interpretation of a machine/network configuration file), and (3) message loading. The arguments lower, upper, and session are used for the first step, config_file and config_tag for the second step, and msg_log_file for the third step. The second step and the third step are not essential and can be skipped. More specifically, the second step and the third step are skipped if config_file and msg_log_file are NULL respectively.

lower and upper respectively specify the lower bound and upper bound of a virtual node space that will be used by a program. A program is allowed to access (e.g., assume, release) only virtual nodes inside the specified range [lower, upper).

session is used for restricting communication among processes. More specifically, session specifies a session name, a null-terminated string determined by a user. The runtime aborts if it receives a message from a remote process which has a different session name. If a user does not need such restriction, session can be NULL. If session is NULL, the runtime does not care the difference of session names.

Note: The concept of *session* has been introduced to detect 'cross talk' among different programs caused by a user's mistake. The 'cross talk' may occur when different programs share the same communication port numbers on the same hosts. Such mistakes can be detected if a user assigns a unique session name to each program. □

config_file specifies a machine/network configuration file that basically contains information about IP numbers (or hostnames) of available machines and port

numbers. The caller process retrieves the information from the file to set up its contact points. If `config_file` is `NULL`, no contact points are set up.

`config_tag` allows the caller process to explicitly control the information that the process retrieves from the file. `config_tag` is `NULL` if the caller does not require such explicit control. Examples of cases when a programmer need to specify `config_tag` are:

- Among many listen ports listed in the file, the programmer would like to explicitly specify the port to which the caller process listens.
- Among many contact points listed in the file, the programmer would like to explicitly specify some of the contact points where the caller process connects.

The details of the syntax of the machine/network configuration file and how `config_tag` is used are described in Section A.7.

`msg_log_file` specifies a log that contains information about flying messages. If `msg_log_file` is `NULL` (this is usually OK), nothing is done. If `msg_log_file` is not `NULL`, `ph_init` function reads `msg_log_file` and restores messages into the local message queue. See also the description of `ph_finalize`.

Note: `lower` and `upper` must be greater than or equal to zero and must be smaller than 2^{63}. □

`void ph_finalize (const char *msg_log_file, int timeout)`

`ph_finalize(msg_log_file, timeout)` function finalizes the Phoenix library.

Before closing communication sockets and disposing data structures, `ph_finalize` makes efforts to prevent flying messages (in this context, they are messages that happen to be in the local message queue of the caller process) from being lost. The arguments of this function are used for this purpose.

This function waits until `timeout` seconds passes or the caller's message queue becomes empty. `ph_finalize` just returns if the message queue becomes empty before timeout. If `timeout` seconds has passed, `ph_finalize` saves contents of messages in the local message queue to `msg_log_file` and returns. Afterward, those messages should be restored in the invocation of `ph_init` by other processes. If `msg_log_file` is `NULL`, flying messages are not saved; those messages may be lost.

In addition to `ph_init` and `ph_finalize`, the Phoenix library provides low-level primitives for the initialization and finalization. While it is complex for a beginner to write the initialization and finalization process with these primitives, they allow more flexible control than `ph_init` and `ph_finalize`. The details of the primitives are described in Section A.6.

A.3 Virtual Node Name Mapping Functions

This section describes functions used for mapping virtual nodes to processes and for accessing currently assumed virtual nodes.

void ph_assume_vps (const ph_vps_t vps)

`ph_assume_vps(vps)` lets a caller process assume virtual nodes `vps`. After the end of function invocation, messages destined for a virtual node in `vps` are delivered to the caller process.

Note: Virtual nodes that have already assumed by the previous call of `ph_assume_vps` are still assumed. For example, suppose that a caller process assuming S calls `ph_assume_vps`(T). Then, the caller assumes $S \cup T$ when the function returns. □

void ph_assume_vp (*ph_vp_t vp*)

`ph_assume_vp(vp)` lets a caller process assume single virtual node `vp`.

void ph_release_vps (*const ph_vps_t vps*)

`ph_release_vps(vps)` lets a caller process release virtual nodes `vps` from its assumed virtual node sets. After the function returns the caller does not receive any messages destined for a virtual node in `vps`.

void ph_release_vp (*ph_vp_t vp*)

`ph_release_vp(vp)` lets a caller process release single virtual node `vp`.

ph_vps_t ph_get_assumed_vps (*void*)

`ph_get_assumed_vps()` returns a set of virtual nodes that a caller process currently assumes. Since the returned data structure is newly allocated in heap space by the runtime system, `ph_vps_destroy` needs to be invoked for freeing the allocated memory region.

Note: A virtual node set returned by this function does not contain the caller process's resource name. □

ph_vp_t ph_get_resource_name (*void*)

`ph_get_resource_name()` returns the *resource name* of the caller process. The resource name is a special virtual node that is randomly chosen by the Phoenix library for each process. It is outside of the range [`ph_get_vp_lower_bound()`, `ph_get_vp_upper_bound()`), and remains constant from the invocation of `ph_init` to the invocation of `ph_finalize`.

The resource name will be useful for the migration protocol that involves processes that have no 'ordinary' virtual nodes [TKEY03].

ph_vp_t ph_get_vp_lower_bound (*void*)

`ph_get_vp_lower_bound()` returns the lower bound of a virtual node space that an application is allowed to use. It is specified by the first argument of the `ph_init` function.

Note: `ph_get_vp_lower_bound() = PH_VP_LOWER_BOUND` holds. □

ph_vp_t ph_get_vp_upper_bound (*void*)

`ph_get_vp_upper_bound()` returns the upper bound of a virtual node space that an application is allowed to use. It is specified by the second argument of the `ph_init` function.

Note: `ph_get_vp_upper_bound() = PH_VP_UPPER_BOUND` holds. □

ph_vp_t ph_get_random_vp (*void*)

`ph_vps_get_random_vp()` returns a virtual node chosen randomly from a virtual node space that an application is allowed to use (i.e., inside [`ph_get_vp_lower_bound()`, `ph_get_vp_upper_bound()`]).

A.4 Message Transmission Functions

This section describes functions for sending and receiving messages.

void ph_send (*ph_vp_t dest, const void *body, size_t len, int tag*)

`ph_send(dest, body, len, tag)` transmits a message to a specified virtual node.

`dest` specifies the destination address of the message. The runtime tries to deliver the message to a physical node that currently assumes `dest`.

`body` and `len` specify the address and the length in bytes of the message respectively.

`tag` is an integer used for message matching. A receive operation specifying a tag will complete successfully only when a message sent with a matching tag arrives. This allows a programmer to deal with the arrival of messages in an orderly way, even if the arrival of messages is not in the order way. `PH_MSG_ANY_TAG` indicates a wild card. A message sent with `PH_MSG_ANY_TAG` will be received with any tag.

Note: This function is *non-blocking*; it does not wait until the message is transmitted to the specified destination. If the runtime finds that no physical node assumes `dest` at the sending time, the message is enqueued to the caller process's local message queue. It remains in the queue until `dest` is assumed by some physical node. □

Caution: Phoenix does not strictly preserve the order of message delivery. The correct order may become broken due to dynamic changes of network topologies. For example, if an application split large data into some pieces and send them one after another by repeating `ph_send`, the pieces of the data may arrive in out of order. A programmer should write applications that do not rely on such property, or manually care the order of message delivery by herself/himself. □

void ph_timed_send (*ph_vp_t dest, const void *body, size_t len, int tag, long long timeout*)

The `ph_timed_send` function is same as the `ph_send` function except that a message sent by `ph_timed_send` automatically disappears if the message is not received by any process during `timeout` micro-seconds.

ph_msg_t ph_recv (*int tag*)

`ph_recv(tag)` receives a message destined for a caller process and returns it. First, this function looks up the caller's local message queue containing messages that have already arrived at the caller. Then, it dequeues some message that satisfies the both of the following conditions:

- The destination virtual node of the message is assumed by the caller.
- The tag of the message matches with `tag`.

As the second condition indicates, `tag` is used for message matching. A receive operation specifying a tag will complete successfully only when a message sent with a matching tag arrives. A receive operation with `PH_MSG_ANY_TAG` can receive a message sent with any tag. This function returns `PH_MSG_INVALID` if an error occurred.

Note: `ph_recv()` is a *blocking* operation; control is not returned to a programmer until a message to be received arrives. □

Note: The message returned by the function is automatically allocated in heap space by the runtime system. Make sure that it is eventually freed by the `ph_msg_destroy` function. □

void ph_msg_destroy (*ph_msg_t msg*)

`ph_msg_destroy(msg)` frees the data structure which `msg` points to.

ph_msg_t ph_try_recv (*int tag*)

The `ph_try_recv` function is same as the `ph_recv` function except that `ph_try_recv` is a *non-blocking* operation. If there exists no messages destined for a caller process, `ph_try_recv` returns `NULL` immediately without waiting the arrival of new messages.

ph_msg_t ph_timed_recv (*int tag, long long timeout*)

The ph_timed_recv function is same as the ph_recv function except that ph_timed_recv returns NULL if no message can be received during timeout micro-seconds.

A typical example of message receive routine is:

```
ph_msg_t msg;

/* Receive a message with any tag */
msg = ph_recv(PH_MSG_ANY_TAG);

/* Do something with the message by
   accessing msg->body, msg->len,
   msg->dest, and msg->tag) */

/* Free the allocated space for the message */
ph_msg_destroy(msg);
```

A.5 Virtual Nodes Manipulation Functions

This section describes functions for dealing with virtual nodes and virtual node sets. These functions includes (i) creation and destruction of virtual node sets, (ii) judgment, and (iii) iterator. The creation (destruction) functions allocate (free) an opaque data structure that represents a set of virtual nodes to (from) head space. The judgment functions are used for comparing two sets or checking properties of a set. The iterator functions are used for accessing a set of virtual nodes iteratively.

The basic data types used by these functions are ph_vp_t, ph_vps_t, and ph_vps_iter_t. ph_vp_t, which denotes a single virtual node, is a 64-bit integer type. ph_vps_t, which denotes a set of virtual nodes, is a pointer type that refers a opaque data structure allocated in heap space. ph_vps_iter_t is used as the type of the iterator.

A.5.1 Creation and Destruction

The Phoenix library provides several functions for creating data structures that denote virtual node sets. You can create singleton sets, interval sets, a union of two sets, and so on.

Note that the newly created data structures are allocated in a heap space; you need to invoke the ph_vps_destroy function to free data structures.

ph_vps_t ph_vps_create_empty (*void*)
ph_vps_create_empty() creates an empty set and returns it.

ph_vps_t ph_vps_create_singleton (*ph_vp_t vp*)
ph_vps_create_singleton(vp) creates a set of virtual nodes that contains only vp and returns it.

ph_vps_t ph_vps_create_range (*ph_vp_t l, ph_vp_t u*)
ph_vps_create_range(l, u) creates an interval of virtual nodes $[\mathtt{l}, \mathtt{u})$ and returns it.

ph_vps_t ph_vps_create_inter (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_create_inter(x, y) creates an intersection set of x and y (i.e., $\{x \cap y\}$) and returns it.

ph_vps_t ph_vps_create_union (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_create_union(x, y) creates a union set of x and y (i.e., $\{x \cup y\}$) and returns it.

ph_vps_t ph_vps_create_diff (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_create_diff(x, y) creates a set difference of x from y (i.e., $\mathtt{x} \setminus \mathtt{y}$) and returns it.

ph_vps_t ph_vps_dup (*const ph_vps_t vps*)
ph_vps_dup(vps) creates a copy of vps and returns it.

void ph_vps_destroy (*ph_vps_t vps*)
ph_vps_destroy(vps) frees a memory region where vps is allocated.

A.5.2 Judgment

int ph_vps_is_empty (*const ph_vps_t vps*)
ph_vps_is_empty(vps) returns 1 if vps is an empty set; otherwise returns 0.

int ph_vps_is_sub (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_is_sub(x, y) returns 1 if x is a subset of y; otherwise returns 0.

int ph_vps_is_equal (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_is_equal(x, y) returns 1 if x denotes an equal set to y; otherwise returns 0.

int ph_vps_is_exclusive (*const ph_vps_t x, const ph_vps_t y*)
ph_vps_is_exclusive(x, y) returns 1 if x and y are exclusive; otherwise returns 0.

int ph_vps_is_member (*const ph_vps_t* vps, *ph_vp_t* vp)

`ph_vps_is_member(vps, vp)` returns `1` if `vp` is a member of `vps`; otherwise returns `0`.

A.5.3 Iterator

The following functions are for accessing virtual node sets iteratively.

ph_vps_iter_t ph_vps_iter_create (*ph_vps_t* vps)

`ph_vps_iter_create(vps)` initializes a iterator over the virtual node set `vps`.

void ph_vps_iter_destroy (*ph_vps_iter_t* x)

`ph_vps_iter_destroy(x)` frees `x`.

int ph_vps_iter_has_next (*ph_vps_iter_t* x)

`ph_vps_iter_has_next(x)` returns `1` if there are some element that has not yet been chosen by the previous calls of `ph_vps_iter_next(x, ...)`; otherwise returns `0`.

void ph_vps_iter_next (*ph_vps_iter_t* x, *ph_vp_t* * lower, *ph_vp_t* * upper, *ph_vp_t* * stride)

`ph_vps_iter_next(x, &l, &u, &s)` gets the *next slice* in the iterator `x`, and stores the lower, upper and stride of the slice to `l`, `u`, and `s` respectively. The function stores `PH_INVALID_VP` to `l`, `u`, and `s` if all the elements have already been chosen by the previous calls of `ph_vps_iter_next(x, ...)`.

More specifically, given lower l, upper u, and stride s, slice $S(l, u, s)$ is defined as follows:

$$S(l, u, s) = \{\ x \mid x = si + l,\ i \in [0, (u - l)/s)\ \}$$

This function gets any slice that is disjoint from all the slices chosen by the previous calls of `ph_vps_iter_next(x, ...)`. In the current version of the library, the function chooses a next slice as follows. Let T be a set of virtual nodes that have not yet been chosen. Let e_1 be the smallest element in T, and e_2 the second smallest element in T if exists.

If $|T| = 1$, the function returns $S(e_1, e_1, 0)$.

If $|T| > 1$, the function returns the largest slice U that satisfies the following condition:

$$\exists e \in T,\ U = S(e_1, e, e_2 - e_1) \wedge U \subseteq T$$

void ph_vps_iter_range_next (*ph_vps_iter_t x, ph_vp_t * lower, ph_vp_t * upper*)
The ph_vps_iter_range_next function is same as the ph_vps_iter_next function except that ph_vps_iter_range_next returns a slice of which stride is 1.

Here is an example of iterator usage. This code prints all the elements in a virtual node set vps.

```
ph_vps_iter_t iter = ph_vps_iter_create(vps);
while (ph_vps_iter_has_next(iter)) {
    ph_vp_t lower, upper, stride, v;
    ph_vps_iter_next(iter, &lower, &upper, &stride);
    for (v = lower; v < upper; v += stride) {
        printf("%lld\n", v);
    }
}
ph_vps_iter_destroy(iter);
```

A.5.4 Miscellaneous

ph_vp_t ph_vps_get_min_elem (*const ph_vps_t vps*)
ph_vps_get_min_elem(vps) returns the smallest element in the virtual node set vps. The function returns PH_INVALID_VP if vps is empty.

ph_vp_t ph_vps_get_max_elem (*const ph_vps_t vps*)
ph_vps_get_max_elem(vps) returns the largest element in vps. The function returns PH_INVALID_VP if vps is empty.

ph_vp_t ph_vps_get_any_elem (*const ph_vps_t vps*)
ph_vps_get_any_elem(vps) randomly chooses one element from vps and returns it. The function returns PH_INVALID_VP if vps is empty.

ph_vp_t ph_vps_get_lower_bound (*const ph_vps_t vps*)
ph_vps_get_lower_bound(vps) returns the smallest element in vps. The function returns PH_INVALID_VP if vps is empty.

Note: ph_vps_get_lower_bound(vps) = ph_vps_get_min_elem(vps) holds. □.

ph_vp_t ph_vps_get_upper_bound (const ph_vps_t vps)
ph_vps_get_upper_bound(vps) returns the upper bound of vps. The function returns PH_INVALID_VP if vps is empty.

Note: ph_vps_get_upper_bound(vps) = ph_vps_get_max_elem(vps)+1 holds. □.

A.6 Low-Level Interface for Initialization and Finalization

The Phoenix library provides low-level interfaces for the flexible initialization and finalization of the library.

A.6.1 Initialization

A basic initialization process consists of the following steps:

- Call `ph_init_runtime` to initialize the runtime.
- Call `ph_add_port_tcp`, `ph_add_port_ssh`, and `ph_add_port_ssl` to set up contact points where the runtime tries to initiate connections.
- Call `ph_load_msgs` to restore messages that were saved by previous processes (if necessary).

void ph_init_runtime (*ph_vp_t* ***lower***, *ph_vp_t* ***upper***)
`ph_init_runtime` initializes the Phoenix runtime. This function must be invoked before any other Phoenix APIs. `lower` and `upper` specify the range that will used by the program.

void ph_add_port_tcp (*const char* ****hostname***, *int* ***port***)

void ph_add_port_ssh (*const char* ****hostname***, *int* ***port***, *const char* ****user***)

void ph_add_port_ssl (*const char* ****hostname***, *int* ***port***)
These functions inform the runtime of contact points at which the node can access remote nodes. The runtime tries to establish connections to the informed addresses. `ph_add_port_tcp`, `ph_add_port_ssh` and `ph_add_port_ssl` specify a contact point for direct TCP communication, SSH tunneling, and SSL communication, respectively.

The arguments of this function specify a contact point. `hostname` can be a DNS hostname or an IP address. IP addresses must be represented as an IPv4 standard text presentation (e.g., `"192.168.0.1"`); `port` specifies a port number to which a node initiates a connection. `user` (the argument that appears only in `ph_add_port_ssh`) specifies an account name on the specified contact point.

void ph_load_msgs (*const char* ****filename***)
`ph_load_msgs(filename)` loads messages from file `filename` and enqueues them to the local message queue.

void ph_set_listen_port (*int lower, int upper*)
ph_set_listen_port(lower, upper) make a caller process listen to any unused port of which number is bigger than or equal to lower and is less than upper.

void ph_set_session_id (*const char * session*)
ph_set_session_id(session) restricts communication among processes. More specifically, session specifies a session name, a null-terminated string determined by a user. The runtime aborts if it receives a message from a remote process which has a different session name.

void ph_read_config (*char * buf, size_t len, const char * config_tag*)
ph_read_config(buf, len, config_tag) sets up of contact points by interpreting machine/network configuration information. The address and the length in bytes of the configuration information are specifed by body and len respectively. config_tag allows the caller process to explicitly control the information that the process retrieves from buf.

We show a typical example of initialization code.

```
int main() {
    const int PORT = 30000;
    const char * USER = "phnx";  /* username on host2 */

    /* initialize the runtime */
    ph_init_runtime(0LL, 1000LL);
    ph_set_listen_port(PORT, PORT+1);

    /* setup contact points */
    ph_add_port_tcp("192.168.0.1", PORT);
    ph_add_port_tcp("host1", PORT);
    ph_add_port_ssh("host2", PORT, USER);

    /* ... */
```

In the above code, the runtime binds and listens to port 30000. The contacts points where the runtime initiates connections are:

- 192.168.0.1 and host1 (to port PORT via TCP)
- host2 (to port PORT via SSH with user "phnx")

A.6.2 Finalization

A basic finalization process consists of the following steps:

1. Call `ph_invalidate_routing` to prevent message transmission from remote nodes.

2. Call `ph_mqueue_is_empty` to check whether the local message queue is empty or not (if necessary).

3. Call `ph_store_msgs` to messages that still remain in the message queue (if necessary).

4. Call `ph_finalize_runtime` to shut down the runtime.

void ph_finalize_runtime ()
`ph_finalize_runtime()` closes all the existing connections and shuts down the runtime. This function must be called at the end of the program.

void ph_invalidate_routing (*void*)
`ph_invalidate_routing()` invalidates the routing tables of the local node's neighbors [1]. This function ensures that any message is not transmitted to the local node after the neighbors' routing tables has been invalidated.

This function is basically invoked before `ph_finalize`. Note that invalidating neighbor's routing table requires a certain amount of time. Thus, `ph_finalize` should be called after a certain amount of time has passed.

int ph_mqueue_is_empty (*void*)
`ph_mqueue_is_empty()` returns `1` if the local message queue is empty; otherwise returns `0`.

void ph_store_msgs (*const char *filename*)
`ph_store_msgs(filename)` saves all message in the local message queue at file `filename`. This function allows a node to leave computation even if the node's message queue is not empty. The stored messages needs to be eventually loaded and delivered by `ph_load_msgs`. `filename`.

The permanent leave is typically written as follows:

[1] The local node tells the neighbors that no virtual nodes are reachable via the local node.

```
const int SLEEP_TIME = 1;

ph_invalidate_routing();
sleep(SLEEP_TIME);

/* wait until the local message queue becomes empty */
while (!ph_mqueue_is_empty()) {
    sleep(A_CERTAIN_AMOUNT_OF_TIME);
}
ph_finalize_runtime();
```

First, by calling `ph_invalidate_routing`, the process tries not to receive further messages from its neighbors. The process sleeps a certain period of time to wait until neighbors' routing tables are completely invalidated. Then, the process waits until messages destined for other processes are delivered from the local message queue to remote hosts and the message queue eventually becomes empty. This condition can hold if the process assumes no virtual nodes and the entire space of the virtual nodes are assumed by other processes. Finally, the process shuts down the runtime by calling `ph_finalize_runtime`.

The following code is a typical example of temporal leave:

```
ph_invalidate_routing();
sleep(A_CERTAIN_AMOUNT_OF_TIME);

ph_store_msgs(filename);
ph_finalize_runtime();
```

The node leaves the computation after storing messages that remains in the local queue at `filename`.

A.7 Machine/Network Configurations

A machine/network configuration is to set up contact points to which processes listen and connect. The machine/network configuration is essential for the Phoenix library, and a user usually needs to give the configuration to processes. For example, suppose that you would like to run processes in a cluster. In this case, to let the processes communicate with one another with TCP/IP, you need to inform them of nodes where they run and ports to which they listen. Suppose that some processes would like to communicate over SSH or SSL channels. In this case, you need to explicitly specify machines to which the processes establish SSH/SSL connections.

A basic method for setting up contact points is to declare them in a configuration file and to give this information to processes. Interpreting a configuration file specified by the third argument of `ph_init`, the processes set up their contact points. In addition to such simple interpretation, contact points declared in one configuration file can be selectively given to each process, depending on the value of `config_tag`, the fourth argument of `ph_init`.

In a configuration file, keyword `listen_port`, `dest`, and `match` are used for the above-mentioned contact point declaration. More specifically, a line that starts with `listen_port` declares a local end point to which a caller process listens. A line that starts with `dest` declares an end point to which a caller process connects and protocol that the process uses. `match` is used for pattern-matching. The pattern-matching allows a caller process to selectively validate contact points declared in one configuration file.

The rest of this section is organized as follows. First, we intuitively explain how `listen_port`, `dest` and `match` are used for contact point declaration through several examples. Next, we show typical examples of machine/network configuration files.

A.7.1 Syntax and Semantics

Keyword `listen_port`

A line that starts with `listen_port` followed by a port number is a declaration of a local end point to which a process listens. For example, the following declaration lets a process listen to port 30000.

```
listen_port 30000
```

You can declare more than one listen ports and let a process listen to any un-used port among the declared ports. This feature allows more than one processes to run on the same machine. A basic way for declaring multiple listen ports is to specify its range as follows:

```
listen_port [lower-upper]
```

A process that the above declaration is given listens to any un-used port of which number is bigger than or equal to *lower*, and is less than *upper*. Here is an example of declaration that lets a process listen to any un-used port in $[30000, 30010)$.

```
listen_port [30000-30010]
```

The notation `[lower-upper]` can appear any times in any position of one local end point declaration. For example, the following line declares listen ports 30000, 30001, 40000, and 40001.

```
listen_port [3-5]00[0-2]
```

Keyword `dest`

A line that starts with `dest` followed by an address and a port number is a declaration of an end point to which a process connects. For example, the following declaration lets a process connect to port 30000 on `host.domain.com`.

```
dest host.domain.com:30000
```

Of course, a target machine can be specified by an IP address (represented as an IPv4 standard text) instead of a DNS hostname.

```
dest 192.168.0.1:30000
```

To declare contact points to which a process connects over SSH/SSL channels, you need to add keyword `ssh/ssl` to the end of the declaration. For example, the following declaration lets a process establish SSH connection to port 30000 on.

```
dest host.domain.com:30000 ssh
```

If an account name of a target machine differs from that of a local machine, the target machine's account name must be specified as follows:

```
dest host.domain.com:30000 ssh user
```

There are basically two methods for declaring multiple contact points. One method is to simply list contact points. For example, the following four lines declare `192.168.0.1:30000`, `192.168.0.1:30001`, `192.168.0.2:30000`, and `192.168.0.1:30001`.

```
dest 192.168.0.1:30000
dest 192.168.0.1:30001
dest 192.168.0.2:30000
dest 192.168.0.1:30001
```

The other method is to specify a range of machine addresses and port numbers. For example, the above four contact points can be declared in one line as follows:

```
dest 192.168.0.[1-3]:[30000-30002]
```

It must be noted that the number of digits can be fixed as follows:

```
dest node[00-64]:30000
```

This declares `node00:30000`, `node01:30000`, ... and `node63:30000`.

Keyword `match`

A basic syntax of pattern-matching code is as follows:

- The top line of pattern-matching code is `match begin`.
- The bottom line of the code is `end`.
- In a region surrounded by `match begin` and `end`, patterns and declarations are listed.

The patter-matching code allows a process to use some of contact points declared in one configuration file. More specifically, declarations listed in the pattern-matching code are validated and are given to a process only if a corresponding pattern matches with `config_tag`, the fourth (second) argument of `ph_init`.

We would like to explain the details of the pattern-matching through several examples. Suppose that there are three processes X, Y, and Z which require the following machine/network configuration:

- X listens to port 30000 and connects to port 30001 on `host.domain.com` over a SSH channel.
- Y listens to port 30001 and connects to port 30002 on its local host.
- Z listens to port 30002 and connects to port 30001 on its local host.

Let `config_tag` of X, Y, and Z be `x`, `y`, and `z` respectively. The pattern-matching code for the above configuration is written as follows:

```
match begin
  x -> listen_port 30000
       dest host.domain.com:30001 ssh
| y -> listen_port 30001
       dest localhost:30002
| z -> listen_port 30002
       dest localhost:30001
end
```

`x`, `y`, and `z` located at the left side of the arrows are patterns. If one of these patterns matches with `config_tag`, corresponding declarations located at the right side of the arrow are validated. For example, for process X of which `config_tag` is `x`, `listen_port 30000` and `dest host.domain.com:30001 ssh` are valid contact point declarations.

A pattern can represent a range as follows:

```
match begin
    n[00-16] -> listen_port 30000
end
```

In the above example, the pattern matches with either `n00`, ... or `n16`.

Note that the pattern-matching is *first match*. It proceeds from the top line to the bottom line; and once a pattern matches with `config_tag`, the rest of the lines below are ignored.

An underline sign (_) matches with any `config_tag`. For the following example, `dest localhost:30000` becomes valid regardless of `config_tag` (as long as no other patterns located above _ match with `config_tag`).

```
match begin
    ...
|  _ -> dest localhost:30000
end
```

The pattern-matching allows a single pattern that represents some range to change corresponding declarations, depending on `config_tag`. For example, the following code lets processes of which `config_tag` is `n00`, ... and `n15` to listen to port 30000, ... and 300015 respectively.

```
match begin
   n[00-16/x] -> listen_port 300%x
end
```

Intuitively, pattern `n[00-16/x]` means that:

- The pattern matches with `n00`, ... and `n15`
- If the pattern matches with `config_tag` s, v is bound to variable `x` such that $v \in \{00, ...15\}$ and `n`$v = s$.

In corresponding declarations, v is substituted to `%x`. For example, if `config_tag` is `n05`, `05` is bound to `x`, and `300%x` is evaluated to `30005`.

Note: Any line that begins with a hash sign (`#`) is a comment: the rest of the line is ignored. □

Note: Keywords can be capitalized. For example, both `listen_port` and `LISTEN_PORT` are OK. □

The complete syntax of machine/network configuration files is shown in Figure A.1.

A.7.2 Examples of Configurations

Example I Here is an example of machine/network configurations in which two processes listen to port 30000 and 30001 on the same local machine, and they establish TCP connections to each other.

```
listen_port [30000-30002]
dest        localhost:[30000-30002]
```

Example II Here is an example of machine/network configurations for a cluster. 128 processes running on `node000.domain.com`, ... and `node127.domain.com` communicate with one another at port 30000.

```
listen_port 30000
dest        node[000-128].domain.com:30000
```

$$
\begin{array}{rcl}
a & ::= & \texttt{a} \mid \ldots \mid \texttt{Z} \\
d & ::= & \texttt{0} \mid \ldots \mid \texttt{9} \\
c & ::= & d \mid a \mid . \mid - \mid \_ \\
nat & ::= & d_1 \ldots d_n \\
str & ::= & c_1 \ldots c_n \\
config & ::= & config\_sub_1 \\
 & & \ldots \\
 & & config\_sub_n \\
config\_sub & ::= & decl \\
 & \mid & \texttt{match begin} \\
 & & \quad pattern_1 -> decl_{11} \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad decl_{1m(1)} \\
 & & \quad \ldots \\
 & & \mid pattern_n -> decl_{n1} \\
 & & \qquad\qquad \ldots \\
 & & \qquad\qquad decl_{nm(n)} \\
 & & \texttt{end} \\
decl & ::= & \texttt{listen\_port}\ nat\_set \mid \texttt{dest}\ str\_set{:}nat\_set\ \{\ prot\ \} \\
nat\_set & ::= & n_1 \ldots n_m \\
n & ::= & d \mid [nat_1 - nat_2] \mid \%a \\
str\_set & ::= & s_1 \ldots s_n \\
s & ::= & c \mid [nat_1 - nat_2] \mid \%a \\
prot & ::= & \texttt{ssh}\ \{\ str\ \} \mid \texttt{ssl}\ str_1\ str_2 \\
pattern & ::= & p_1 \ldots p_n \mid \_ \\
p & ::= & c \mid [nat_1 - nat_2] \mid [nat_1 - nat_2/a]
\end{array}
$$

Figure A.1: Syntax of configuration files

Example III Here is an example of machine/network configurations for a cluster, each of which node is a multi-processor machine. 256 processes running on `node000.domain.com`, ... and `node127.domain.com` communicate with one another at port 30000 and 30001 (two processes per one node).

```
listen_port[30000-30002]
dest        node[000-128].domain.com:[30000-30002]
```

Example IV Suppose that you would like to run processes on two different clusters. One cluster consists of node `x000`, ... and `x127`. The other cluster consists of node `y00`, ... and `y63`. The inter-cluster communication is not allowed except a SSH channel from `x000` to `y00`.

Let `config_tag` of each process be equal to a hostname where the process runs. In this case, a machine/network configuration can be written as follows:

```
listen_port 30000
match begin
  x000       -> dest y00:30000 ssh user
                dest x[000-128]:30000
| x[001-128] -> dest x[000-128]:30000
| y[00-64]   -> dest y[00-64]:30000
end
```

Note: In contrast to common MPI implementation, the Phoenix library supports resource discovery mechanism for facilitating ease of configuration. This mechanism allows a user not to declare all possible contact points and to specify only essential information for communication between processes. See Chapter 3 for the details. □

Appendix B

Sample Programs Written in Phoenix

This chapter presents two sample programs written in Phoenix. Section B.1 describe a short program in which one process sends a message to another, and explains how the Phoenix APIs work. Section B.2 mention how programs that support join and leave of nodes are written in Phoenix.

B.1 A Short Example

Here is a example program `hello.c` in which a child process transmits string `"HELLO"` to a parent process.

```
 1: #include <stdio.h>
 2: #include <unistd.h>
 3: #include "phnx/phnx.h"
 4:
 5: int main(void) {
 6:     pid_t pid;
 7:     ph_vps_t vps;
 8:     ph_msg_t msg;
 9:
10:     pid = fork();
11:
12:     /*** Initialization ***/
13:     ph_init(0LL, 32LL, "machines", NULL, NULL, NULL);
14:
15:     if (pid == 0) {
16:         /*** S E N D E R ***/
17:
18:         /*** Node name mapping ***/
```

```
19:         vps = ph_vps_create_range(0LL, 16LL);
20:         ph_assume_vps(vps);
21:         ph_vps_destroy(vps);
22:
23:         /*** Message transmission ***/
24:         ph_send(24LL, "HELLO", 6, PH_MSG_ANY_TAG);
25:     } else {
26:         /*** R E C E I V E R ***/
27:
28:         /*** Node name mapping ***/
29:         vps = ph_vps_create_range(16LL, 32LL);
30:         ph_assume_vps(vps);
31:         ph_vps_destroy(vps);
32:
33:         /*** Message receipt ***/
34:         msg = ph_recv(PH_MSG_ANY_TAG);
35:         printf("%s\n", (char *)msg->body);
36:         ph_msg_destroy(msg);
37:     }
38:
39:     /*** Finalization ***/
40:     ph_finalize(NULL, -1);
41:
42:     return 0;
43: }
```

As mentioned earlier, in Phoenix, message destinations are specified by virtual nodes, which are dynamically mapped to processes. Thus, to send a message to the parent process, the child process needs to know virtual nodes mapped to the parent process and specify the message destination with the virtual nodes. In this program,

- the child process assumes virtual node names $[0, 16)$;
- the parent process assumes virtual node names $[16, 32)$; and
- the child process transmits `"HELLO"` to virtual node 24, which is assumed by the parent process.

To execute this program, you need to create a configuration file that specifies available machines/networks, and give it to the program. The Phoenix runtime basically binds and listens to a port specified in the configuration file. For example, if you would like to run a program at port 30000 and 30001 on the local host, a configuration file can be written as follows:

```
listen_port  [30000-30002]
dest         localhost:[30000-30002]
```

We name this configuration file `machines` and assume that the file is located on the same directory where `hello.c` is located.

After writing the configuration file, you can compile `hello.c` and execute generated executable code. When you execute it, the child process sends `"HELLO"` to the parent process, and then the parent prints a received string (i.e., `"HELLO"`) on a terminal.

```
% gcc  -lphnx -lpthread -I/usr/local/include -L/usr/local/lib \
       -o hello hello.c
% ./hello
HELLO
```

In the rest of this section, we take a closer look at program `hello.c` and configuration file `machines`.

B.1.1 A Closer Look at `hello.c`

We explain some of the Phoenix APIs that appear in `hello.c`.

Header File The following line of code includes the common header file for Phoenix APIs.

```
 3: #include "phnx/phnx.h"
```

A programmer must include `phnx/phnx.h` into her/his program to make the API functions available.

Variable Definitions The following lines define variables that are used in the function `main`.

```
 7:     ph_vps_t vps;
 8:     ph_msg_t msg;
```

`ph_vps_t` is a type for a set of virtual node names. `vps` is used as an argument of virtual node name mapping function (`ph_assume_vps`).

`ph_msg_t` is a type for a message. `msg` is used for storing a return value of message receive function `ph_receive`.

Initialization The following line of code initializes the Phoenix runtime.

```
13:     ph_init(0LL, 32LL, "machines", NULL, NULL, NULL);
```

The function `ph_init` is an initialization function for opening the underlying communication layer on top of which Phoenix provides the simpler name space and message delivery semantics. This function must be invoked before any other library functions.

The first (second) argument `0LL` (`32LL`) specifies the lower (upper) bound of a virtual node name space that is used by this program. The program can only use a virtual node name in this specified range $[0, 32)$. To use any virtual node names outside the range is prohibited by the Phoenix runtime.

`machines` denotes the name of a configuration file. The runtime uses the information obtained by the configuration file to (i) determines a local end point to which the runtime listens and (ii) remote end points to which the runtime tries to connect. The runtime use the established connections to route messages between any pair of nodes, not just between its end points.

Node Name Mapping The following lines map virtual node names to physical nodes.

```
19:         vps = ph_vps_create_range(0LL, 16LL);
20:         ph_assume_vps(vps);
21:         ph_vps_destroy(vps);

29:         vps = ph_vps_create_range(16LL, 32LL);
30:         ph_assume_vps(vps);
31:         ph_vps_destroy(vps);
```

The `ph_vps_create_range` function returns an interval of virtual nodes. In this sample program, `ph_vps_create_range(0LL, 16LL)` `ph_vps_create_range(16LL, 32LL)` return virtual nodes $[0, 16)$ and $[16, 32)$ respectively. A data structure returned by this function is allocated on heap space. The `ph_vps_destroy` function is invoked to free the allocated memory region.

The `ph_assume_vps` function maps virtual nodes to a caller process. When process P calls `ph_assume_vps(vps)`, the Phoenix runtime system starts delivering messages destined for virtual nodes in `vps` to P. In other words, this is the caller's declaration that it is ready for receiving messages destined for virtual nodes in `vps`.

In `hello.c`, the child process assumes $[0, 16)$ (at line 18). The parent process assumes $[16, 32)$ (at line 28).

Note: The `ph_release_vps` function has the reverse effect. The system no longer delivers messages destined for `vps` to the caller node. □

Message Transmission/Receipt The following line of code transmits `"HELLO"` to virtual node 24.

```
24:         ph_send(24LL, "HELLO", 6, PH_MSG_ANY_TAG);
```

The first argument `24LL` specifies the destination address of the message. The runtime tries to deliver the message to a process that currently assumes 24. If no processes

assume 24, the message is enqueued to the process's local queue. The message remains in the queue until the destination virtual node is assumed by some process.

The second argument "HELLO" and the third argument 6 specify the address and the length in bytes of the message respectively.

The fourth argument is a tag used for message matching. PH_MSG_ANY_TAG indicates a wild card; messages sent with PH_MSG_ANY_TAG will be received with any tag.

The following line of code receives any message destined for the local virtual nodes.

```
34:         msg = ph_recv(PH_MSG_ANY_TAG);
```

The argument of this function is used for message matching. A receive operation with PH_MSG_ANY_TAG can receive a message sent with any tag.

The ph_recv function returns a pointer to a structure that has the following fields:

- body: pointer to the body of the message
- len: length in bytes of the body
- dest: message destination
- tag: message tag

In this case, when the parent receives the message msg, msg->body, msg->len, msg->dest, and msg->tag are "HELLO", 6, 24, and PH_MSG_ANY_TAG respectively.

Note: The returned data structure is allocated on a heap space by the runtime. □

The following line of code frees msg to the message allocated on the receipt of the message.

```
36:         ph_msg_destroy(msg);
```

Finalization The following line of code shuts down the runtime system and allows the node to leave computation.

```
40:     ph_finalize(NULL, -1);
```

This function must be called at the end of the program.

B.1.2 A Closer Look at `machines`

Here, again, is configuration file machines.

```
listen_port  [30000-30002]
dest         localhost:[30000-30002]
```

As mentioned earlier, the configuration informs the Phoenix runtime of available machines/networks. The first line specifies a local end point of the underlying communication layer. The second line specifies hostnames/IP addresses to which the runtime tries to maintain connections.

In this example, the child and parent process binds and listens to port 30000 and 30001 respectively or vice verse. And they try to establish a TCP connection to port 30000 and 30001 on the local host (i.e., to each other).

B.2 Writing Programs Supporting Join and Leave of Processes

In this section, we describe how programs that support join and leave of processes are written in Phoenix. Section B.2.1 describes rules of virtual node assignment. Section B.2.2 presents a basic strategy for exchanging virtual nodes among processes. Section B.2.3 shows a sample C program in which processes join and leave dynamically with exchange messages.

B.2.1 Rules of Virtual Node Assignment

As mentioned in Section 2.2.1, programs written in Phoenix must follow the following disjoint-cover property:

- No two processes assume the same virtual node at any instant.
- There may be an instant at which no process assumes a virtual node, but in such cases, one must eventually appear that assumes it.

This property can hold easily if no processes join/leave dynamically. As shown in `hello.c`, a programmer can maintain the property by statically partitioning the space among participating processes. and by assuming nodes are fixed throughout the entire computation.

On the other hand, building applications in which each process can autonomously decide to join/leave computation requires a dynamic protocol to maintain the property. First, the application must be able to transport virtual nodes (i.e., change the mapping between virtual node and processes with satisfying the disjoint-cover property). Second, processes should support migration of application-level states in such a way that the migration becomes transparent from the processes not involved in it.

To see the importance of the second requirement, let us consider an application that partitions a large hash table (or any "container" data structure such as an array) among participating processes. Such an application typically uses a simple mapping between hash keys to *virtual* nodes. Most simply, hash key k is mapped to virtual node k and

lookup/update of an item with hash key k is destined for whichever process assumes virtual node k at that moment.

For such a mechanism to support transparent migration, we must guarantee that a process assuming virtual node k always has all valid items of hash key k. This requires processes to migrate hash table items from one node to another upon migrating virtual nodes. The same situation arises in every application that partitions application-level states among processes.

B.2.2 A Basic Strategy for Exchanging Virtual Nodes

A basic strategy for exchanging virtual nodes are briefly sketched as follows:

- When process P wants to join computation, it asks some process Q to delegate some of its virtual nodes. Specifically, P sends a message to a randomly selected virtual node (that is assumed by some process Q), receives a reply from Q, and assumes virtual nodes that Q gives to P (See Figure B.1).

- When process P wants to leave computation, it asks some processor Q to take over P's virtual nodes.

In both cases, while transporting virtual nodes, P and Q exchange messages to transport application data associated with the virtual nodes if necessary.

Note that a joining/leaving process comes to assume no virtual node names: a joining process first assumes no virtual nodes, and a leaving process also has no virtual nodes after releasing all its virtual nodes. Nevertheless, they clearly need to somehow receive a message. For example, a joining process needs to receive a message to take over virtual nodes. Phoenix needs to allows a process that assumes no virtual nodes to receive a message.

For this purpose, Phoenix provides the `ph_get_resource_name` function. This function returns an opaque virtual node name *outside* the virtual node name space that an application uses. The resulting name is worldwide unique with high probability[1]. It is generated when a process brings up (i.e., calls `ph_init`) and returns the same value until it disconnects from the application (i.e., calls `ph_finalize`). Besides regular messages whose destinations are in the virtual node name space, Phoenix also routes messages destined for such node names. In short, the node name returned by `ph_get_resource_name` serves as the name bound to the process. Thus, we hereby call this name *resource name* of the process.

Caution: Applications should not use resource names for the application-level logic. They are only used for supporting migration. □

[1] We assume it is in fact unique.

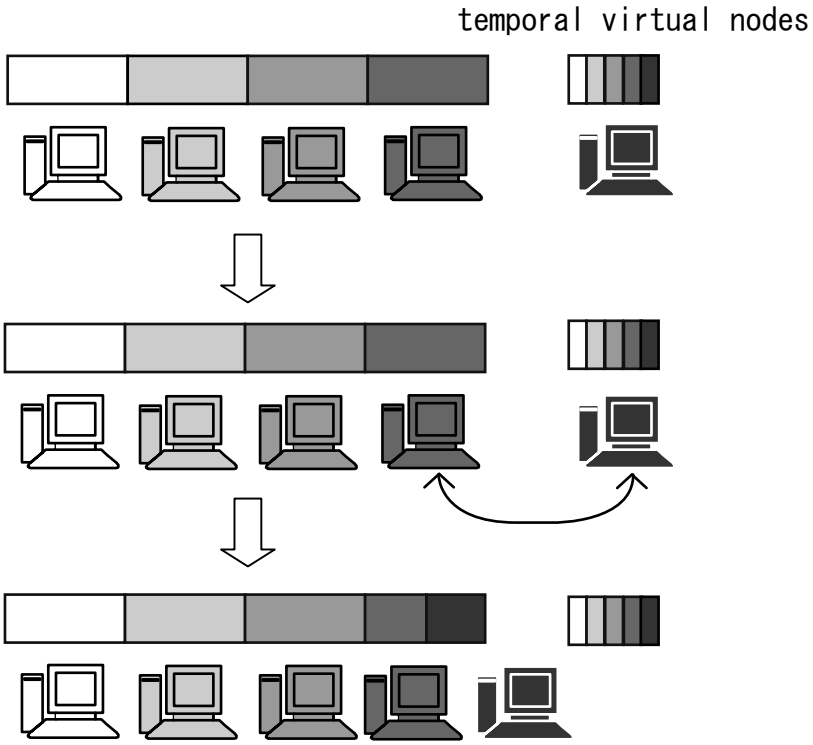

Figure B.1: A basic strategy for a new process to join computation

B.2.3 A Sample Program

We show a example program in which processes continue to forward "HELLO" messages among them while some of them join and leave computation dynamically. In the program, the first forked process called an *initiator* assumes the entire virtual node. Two processes that follow the *initiator* are called *slaves*. The slaves ask a process that has already joined computation for some of its virtual nodes. Each process leaves computation after it receives "HELLO" messages several times. When leaving, it delegates the virtual nodes that it assumes to another process.

More specifically, join and leave are handled as follows:

Join: Suppose that process P wants to join computation.

1. P sends a JOIN_REQUEST message to a randomly chosen virtual node v to ask for virtual nodes. P attaches its resource name r. This resource name is used for the address of the reply of this JOIN_REQUEST message.
2. Process Q that assumes v receives the JOIN_REQUEST message. Q divides its virtual nodes into two parts, and releases either of the parts and gives it to P. Then, Q sends a JOIN_RESPONSE (attached with the given virtual nodes) to P's resource name r.
3. P receives the JOIN_RESPONSE message and assumes the virtual nodes attached with the message.

Leave: Suppose process P wants to leave computation.

1. P releases all its virtual nodes and devolves them to another process in such a way that each node always assumes a single interval (i.e., a contiguous range of integers) of virtual nodes, rather than any kinds of sets[2].

   For this purpose, P sends a LEAVE_REQUEST (attached with P's virtual nodes) to a virtual node adjacent to P's virtual nodes. For example, if P's virtual nodes are $[16, 24)$, it gives its virtual nodes to virtual node 15 or 24.
2. Q receives the LEAVE_REQUEST message and takes over the virtual nodes attached with the message. Then Q sends a LEAVE_RESPONSE message as acknowledgment of the request.
3. P receives the LEAVE_RESPONSE message and exits.

We show an actual C program in which individual process acts as mentioned above. For readability, we divide the program into six fragments. Here is the first fragment of the program.

[2]We believe it is typical for an application to assume a single or a few intervals for the sake of simplicity and the worst case storage requirements

```
 1: #include <stdio.h>
 2: #include <assert.h>
 3: #include <unistd.h>
 4: #include "phnx/phnx.h"
 5:
 6: /* kind of message */
 7: enum kind {
 8:      HELLO, JOIN_REQUEST, JOIN_RESPONSE, LEAVE_REQUEST, LEAVE_RESPONSE
 9: };
10:
11: /* message data structure */
12: struct msg {
13:      enum kind kind;
14:      ph_vp_t   sender;       /* for JOIN_REQUEST and LEAVE_REQUEST */
15:      ph_vp_t   lower, upper; /* for JOIN_RESPONSE and LEAVE_REQUEST */
16: };
17: typedef struct msg * msg_t;
```

The above defines `struct msg` which denotes a message exchanged between processes. `struct msg` has the following fields:

- `kind`: a kind of message.
- `sender`: a resource name of a process that sends this message. It is used for specifying an address to which a response of the message is transmitted. This field is valid only if `kind` is either `JOIN_REQUEST` or `LEAVE_REQUEST`.
- `lower`, `upper`: virtual nodes that migrates due to join/leave of nodes. This field is valid only if `kind` is either `JOIN_RESPONSE` or `LEAVE_RESPONSE`.

Here is the second fragment of the program.

```
18: /* ph_assume_vps() with pretty print */
19: void assume_vps(ph_vps_t vps) {
20:      ph_vps_t old, new;
21:
22:      old = ph_get_assumed_vps();
23:      ph_assume_vps(vps);
24:      new = ph_get_assumed_vps();
25:
26:      printf("\t[%lld, %lld) ---> [%lld, %lld)\n",
27:             ph_vps_get_lower_bound(old), ph_vps_get_upper_bound(old),
28:             ph_vps_get_lower_bound(new), ph_vps_get_upper_bound(new));
29:
30:      ph_vps_destroy(old);
31:      ph_vps_destroy(new);
```

```
32: }
33:
34: /* ph_release_vps() with pretty print */
35: void release_vps(ph_vps_t vps) {
36:      ph_vps_t old, new;
37:
38:      old = ph_get_assumed_vps();
39:      ph_release_vps(vps);
40:      new = ph_get_assumed_vps();
41:
42:      printf("\t[%lld, %lld) ---> [%lld, %lld)\n",
43:             ph_vps_get_lower_bound(old), ph_vps_get_upper_bound(old),
44:             ph_vps_get_lower_bound(new), ph_vps_get_upper_bound(new));
45:
46:      ph_vps_destroy(old);
47:      ph_vps_destroy(new);
48: }
```

`assume_vps(vps)` assumes virtual nodes `vps`. `release_vps(vps)` releases virtual nodes `vps`. When assuming/releasing virtual nodes, these functions print previous and current virtual nodes.

Here is the third fragment of the program.

```
49: void send_join_request(void) {
50:      ph_vp_t    dest;
51:      struct msg m;
52:
53:      dest = ph_get_random_vp();
54:      m.kind = JOIN_REQUEST;
55:      m.sender = ph_get_resource_name();
56:      printf("(%d)\tsends JOIN_REQUEST to %lld.\n", getpid(), dest);
57:      ph_send(dest, (void *)&m, sizeof(struct msg), PH_MSG_ANY_TAG);
58: }
59:
60: void receive_join_response(void) {
61:      ph_vps_t vps;
62:      ph_msg_t x;
63:      msg_t    m;
64:
65:      x = ph_recv(PH_MSG_ANY_TAG);
67:      m = (msg_t)x->body;
68:      assert(m->kind == JOIN_RESPONSE);
69:
70:      printf("(%d)\treceives JOIN_RESPONSE.\n", getpid());
71:      printf("(%d)\ttakes over [%lld, %lld).\n", getpid(),
```

```
 72:             m->lower, m->upper);
 73:      vps = ph_vps_create_range(m->lower, m->upper);
 74:      assume_vps(vps);
 75:      ph_vps_destroy(vps);
 76:      ph_msg_destroy(x);
 77: }
 78:
 78: void join(void) {
 80:      send_join_request();
 81:      receive_join_response();
 82: }
```

The `join` function is invoked to the caller to ask any process for virtual nodes. Here is the fourth fragment of the program.

```
 83: ph_vp_t get_adjacent_vp(ph_vps_t vps) {
 84:      ph_vp_t l = ph_vps_get_lower_bound(vps);
 85:      ph_vp_t u = ph_vps_get_upper_bound(vps);
 86:      return (l == PH_VP_LOWER_BOUND) ? u : (l - 1LL);
 87: }
 88:
 89: void send_leave_request(ph_vp_t dest, ph_vps_t vps) {
 90:      struct msg m;
 91:
 92:      m.kind = LEAVE_REQUEST;
 93:      m.sender = ph_get_resource_name();
 94:      m.lower = ph_vps_get_lower_bound(vps);
 95:      m.upper = ph_vps_get_upper_bound(vps);
 96:      printf("(%d)\tsends LEAVE_REQUEST to %lld.\n", getpid(), dest);
 97:      ph_send(dest, (void *)&m, sizeof(struct msg), PH_MSG_ANY_TAG);
 98: }
 99:
100: void receive_leave_response(void) {
101:      ph_msg_t x;
102:      msg_t    m;
103:
104:      x = ph_recv(PH_MSG_ANY_TAG);
105:      m = (msg_t)x->body;
106:      assert(m->kind == LEAVE_RESPONSE);
107:      printf("(%d)\treceives LEAVE_RESPONSE\n", getpid());
108:      ph_msg_destroy(x);
109: }
110:
111: void leave(void) {
112:      ph_vps_t vps  = ph_get_assumed_vps();
113:      ph_vp_t  dest = get_adjacent_vp(vps);
```

```
114:      printf("(%d)\tgives [%lld, %lld) to %lld.\n", getpid(),
115:             ph_vps_get_lower_bound(vps), ph_vps_get_upper_bound(vps), dest);
116:      release_vps(vps);
117:      send_leave_request(dest, vps);
118:      ph_vps_destroy(vps);
119:      receive_leave_response();
120: }
```

The `leave` function is invoked to devolve the caller's virtual nodes to another process.

Here is the fifth fragment of the program.

```
121: void send_hello(void) {
122:      ph_vp_t    dest = ph_get_random_vp();
123:      struct msg m;
124:
125:      m.kind = HELLO;
126:      printf("(%d)\tsends HELLO to %lld.\n", getpid(), dest);
127:      ph_send(dest, (void *)&m, sizeof(struct msg), PH_MSG_ANY_TAG);
128: }
129:
120: ph_vps_t get_half_of_assumed_vps(void) {
121:      ph_vps_t my_vps = ph_get_assumed_vps();
122:      ph_vp_t  l      = ph_vps_get_lower_bound(my_vps);
123:      ph_vp_t  u      = ph_vps_get_upper_bound(my_vps);
124:      ph_vps_t vps    = ph_vps_create_range(l, l + (u - l) / 2);
125:      ph_vps_destroy(my_vps);
126:      return vps;
127: }
128:
129: void send_join_response(ph_vp_t sender, ph_vps_t vps) {
130:      struct msg m;
131:
132:      m.kind = JOIN_RESPONSE;
133:      m.lower = ph_vps_get_lower_bound(vps);
134:      m.upper = ph_vps_get_upper_bound(vps);
135:      ph_send(sender, (void *)&m, sizeof(struct msg), PH_MSG_ANY_TAG);
136: }
137:
138: void send_leave_response(ph_vp_t sender) {
139:      struct msg m;
140:      m.kind = LEAVE_RESPONSE;
141:      ph_send(sender, (void *)&m, sizeof(struct msg), PH_MSG_ANY_TAG);
142: }
143:
```

```
144: void main_loop(int nhellos) {
145:      int n = 0;
146:      while ((nhellos == -1) || (n < nhellos)) {
147:           ph_msg_t x = ph_recv(PH_MSG_ANY_TAG);
148:           msg_t    m = (msg_t)x->body;
149:           switch (m->kind) {
150:           case HELLO:
151:                printf("(%d)\treceives HELLO.\n", getpid());
152:                sleep(1);
153:                send_hello();
154:                n++;
155:                break;
156:           case JOIN_REQUEST:  {
157:                ph_vps_t vps = get_half_of_assumed_vps();
158:
159:                printf("(%d)\treceives JOIN_REQUEST.\n", getpid());
160:                printf("(%d)\tgives [%lld, %lld) to %lld.\n", getpid(),
161:                       ph_vps_get_lower_bound(vps),
162:                       ph_vps_get_upper_bound(vps), m->sender);
163:                release_vps(vps);
164:                send_join_response(m->sender, vps);
165:                ph_vps_destroy(vps);
166:                break; }
167:           case LEAVE_REQUEST: {
168:                ph_vps_t vps;
169:
170:                printf("(%d)\treceives LEAVE_REQUEST.\n", getpid());
171:                printf("(%d)\ttakes over [%lld, %lld) from %lld.\n",
172:                       getpid(), m->lower, m->upper, m->sender);
173:                vps = ph_vps_create_range(m->lower, m->upper);
174:                assume_vps(vps);
175:                ph_vps_destroy(vps);
176:
177:                send_leave_response(m->sender);
178:                break; }
179:           }
180:           ph_msg_destroy(x);
181:      }
182: }
```

The `main_loop` function is called after the process has joined computation. `main_loop(nhellos)` repeats receiving messages and handling them until it receives `"HELLO"` message `nhellos` times. If `nhellos` is -1, this function repeats receiving messages forever.

Here is the last fragment of the program.

```
183: void spawn_initiator(int nhellos) {
```

```
184:      ph_vps_t vps;
185:
186:      if (fork() != 0) { return; }
187:
188:      ph_init(0LL, 32LL, "machines", NULL, NULL, NULL);
189:      printf("(%d)\tstarts.\n"
190:             "\tresource vp = %lld\n", getpid(), ph_get_resource_name());
191:
192:      /* assume all the virtual node names */
193:      vps = ph_vps_create_range(PH_VP_LOWER_BOUND, PH_VP_UPPER_BOUND);
194:      assume_vps(vps);
195:      ph_vps_destroy(vps);
196:
197:      send_hello();
198:      main_loop(nhellos);
199:      leave();
200:      ph_finalize(NULL, -1);
201:      printf("(%d)\texits.\n", getpid());
202:      exit(0);
203: }
204:
205: void spawn_slave(int nhellos) {
206:      if (fork() != 0) { return; }
207:
208:      ph_init(0LL, 32LL, "machines", NULL, NULL, NULL);
209:      printf("(%d)\tstarts.\n\tresource vp = %lld\n", getpid(),
210           ph_get_resource_name());
211:
212:      join();
213:      main_loop(nhellos);
214:      leave();
215:      ph_finalize(NULL, -1);
216:      printf("(%d)\texits.\n", getpid());
217:      exit(0);
218: }
219:
220: int main(void) {
221:      spawn_initiator(3);
222:      sleep(1);
223:
224:      spawn_slave(3);
225:      sleep(1);
226:
227:      spawn_slave(-1);
228:      sleep(1);
```

```
229:
230:      while (1) { sleep(10); }
231:      return 0;
232: }
```

The function `main` forks three processes by calling `spawn_initiator` and `spawn_slave`.

- A first forked process (initiator) assumes the entire virtual node, receives messages three times, and leave computation by calling `leave`.
- A second forked process joins computation by calling `join`, receives messages three times, and leave computation by calling `leave`.
- A third forked process joins computation by calling `join`, and repeat receiving messages forever.

Here is an example output of the program.

```
(24577) starts.                 // (24577) joins in computation.
        resource vp = 9104594110672513012
        [-1, -1) ---> [0, 64)
(24577) sends HELLO to 16.
(24577) receives HELLO.
(24584) starts.                 // (24584) begins to join in computation.
        resource vp = 6332194092195308135
(24584) sends JOIN_REQUEST to 55.
(24577) sends HELLO to 18.
(24577) receives HELLO.
(24591) starts.                 // (24591) begins to join in computation.
        resource vp = 7645442783267026781
(24591) sends JOIN_REQUEST to 26.
(24577) sends HELLO to 51.
(24577) receives JOIN_REQUEST.
(24577) gives [0, 32) to 6332194092195308135.
        [0, 64) ---> [32, 64)
(24577) receives HELLO.
(24584) receives JOIN_RESPONSE.
(24584) takes over [0, 32).
        [-1, -1) ---> [0, 32)   // (24584) finishes to join in computation.
(24584) receives JOIN_REQUEST.
(24584) gives [0, 16) to 7645442783267026781.
        [0, 32) ---> [16, 32)
(24591) receives JOIN_RESPONSE.
(24591) takes over [0, 16).
        [-1, -1) ---> [0, 16)   // (24591) finishes to join in computation.
```

```
(24577) sends HELLO to 15.
(24577) gives [32, 64) to 31.   // (24577) begins to leave computation.
        [32, 64) ---> [-1, -1)
(24577) sends LEAVE_REQUEST to 31.
(24584) receives LEAVE_REQUEST.
(24584) takes over [32, 64) from 9104594110672513012.
        [16, 32) ---> [16, 64)
(24577) receives LEAVE_RESPONSE
(24591) receives HELLO.
(24591) sends HELLO to 27.
(24584) receives HELLO.
(24577) exits.                  // (24577) finishes to leaves computation.
(24584) sends HELLO to 52.
(24584) receives HELLO.
(24584) sends HELLO to 54.
(24584) receives HELLO.
(24584) sends HELLO to 30.
(24584) gives [16, 64) to 15.
        [16, 64) ---> [-1, -1)
(24584) sends LEAVE_REQUEST to 15.
(24591) receives LEAVE_REQUEST.
(24591) takes over [16, 64) from 6332194092195308135.
        [0, 16) ---> [0, 64)
(24584) receives LEAVE_RESPONSE
(24591) receives HELLO.
(24591) sends HELLO to 10.
(24591) receives HELLO.
(24584) exits.                  // (24584) finishes to leaves computation.
(24591) sends HELLO to 1.
(24591) receives HELLO.
(24591) sends HELLO to 45.
...
```

The process of which process ID (PID) is 24577 first joins computation, and the process with PID 24584 and the one with PID 24591 successively join computation. Then, after receiving several messages the process with PID 24577 and the one with PID 24584 leave computation. As this output indicated, while some of the processes are joining or leaving, the other processes continue to forward `"HELLO"` messages.

Bibliography

[ABB01] David A. Patterson Aaron B. Brown. Embracing Failure: A Case for Recovery-Oriented Computing (ROC). In *Proceedings of the High Performance Transaction Processing Symposium*, October 2001.

[AD03] Ittai Abraham and Danny Dolev. Asynchronous Resource Discovery. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC '03)*, pages 143–150, May 2003.

[Adv05] Advanced Micro Devices. *AMD64 Virtualization Codenamed "Pacifica" Technology Secure Virtual Machine Architecture Reference Manual*, 2005.

[AFT99] Yariv Aridor, Michael Factor, and Avi Teperman. cJVM: a Single System Image of a JVM on a Cluster. In *Proceedings of the International Conference on Parallel Processing (ICPP '99)*, pages 4–11, September 1999.

[AK93] Sudhanshu Aggarwal and Shay Kutten. Time Optimal Self-Stabilizing Spanning Tree Algorithms. In *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 400–410, 1993.

[AKY91] Yehuda Afek, Shay Kutten, and Moti Yung. Memory-efficient Self Stabilizing Protocols for General Networks. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, pages 15–28, 1991.

[AM03] Oliver Aumage and Guillaume Mercier. MPICH/MADIII: a Cluster of Clusters Enabled MPI Implementation. In *Proceedings of the 3th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '03)*, pages 26–33, May 2003.

[Apa] Apache Sofrware Fondation. `http://www.apache.org/`.

[AR04] Amr Awadallah and Mendel Rosenblum. The vMatrix: Server Switching. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS '04)*, pages 110–118, May 2004.

[ATS03] Stephanos Androutsellis-Theotokis and Diomidis Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys (CSUR)*, pages 335–371, December 2003.

[BAG00] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An Architecture for a Resource Management and Scheduling System in a Global Computational Grid. In *Proceedings of the 8th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia '00)*, pages 283–289, May 2000.

[BBB96] J. Eric Baldeschwieler, Robert D. Blumofe, and Eric A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, pages 165–172, September 1996.

[BBC+02] George Bosilca, Aurélien Bouteiller, Franck Cappello, Samir Djilali, Gilles Fédak, Cécile Germain, Thomas Hérault, Pierre Lemarinier, Oleg Lodygensky, Frédéric Magniette, Vincent Néri, and Anton Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, pages 1–18, November 2002.

[BBC+04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating System Support for Planetary-Scale Network Services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 253–266, 2004.

[BCZ90] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-specific Memory Coherence. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Princiles and Practice of Parallel Programming (PPoPP '90)*, pages 168–176, 1990.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP-19)*, pages 164–177, October 2003.

[BDH03] Luiz Barroso, Jeffrey Dean, and Urs Hölzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[BDR97] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. In *Proceedings*

of the 16th ACM Symposium on Operating Systems Principles (SOSP-16), pages 143–156, October 1997.

[BL97] Robert D. Blumofe and Philip A. Lisiecki. Adaptive and Reliable Parallel Computing on Networks of Workstations. In *Proceedings of the USENIX Annual Technical Conference (USENIX '97)*, pages 133–148, January 1997.

[BL98] Amnon Barak and Oren La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing. *Journal of Future Generation Computer Systems (FGCS)*, 13(4-5):361–372, March 1998.

[BNC+01] Rajanikanth Batchu, Jothi P. Neelamegam, Zhenqian Cui, Murali Beddhu, Anthony Skjellum, Yoginder Dandass, and Manoj Apte. MPI/FT™: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '01)*, pages 26–33, 2001.

[Boc] Bochs: The Open Source IA-32 Emulation Project. `http://bochs.sourceforge.net/`.

[BPZ96] Martin Backschat, Alexander Pfaffinger, and Christoph Zenger. Economic-Based Dynamic Load Distribution in Large Workstation Networks. In *Proceedings of the 2nd International European Conference on Parallel Processing, Volume II (Euro-Par '96, Vol. II)*, pages 631–634, 1996.

[BS96] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-based fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 14(1):80–107, 1996.

[Buy99] Rajkumar Buyya. *High Performance Cluster Computing: Architectures and Systems*. Prentice Hall PTR, June 1999.

[CCK+95] Jeremy Casas, Dan L. Clark, Ravi B. Konuru, Steve W. Otto, Robert M. Prouty, and Jonathan Walpole. MPVM: A Migration Transparent Version of PVM. *Computing Systems*, 8(2):171–216, Spring 1995.

[CDD+05] Franck Cappello, Frederic Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jegou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000: A Large Scale, Reconfigurable, Controlable and Monitorable Grid Platform. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, pages 99–106, November 2005.

[CDF+05] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, and Vincent Néri et Oleg Lodygensky. Computing on Large Scale Distributed Systems: XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. *Journal of Future Generation Computer Systems (FGCS)*, 21(3):417–437, March 2005.

[CFH+05] Christopher Clark, Keir Fraser, Steven Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, May 2005.

[CFK+98] Karl Czajkowski, Ian T. Foster, Nicholas T. Karonis, Carl Kesselman, Stuart Martin, Warren Smith, and Steven Tuecke. A Resource Management Architecture for Metacomputing Systems. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[CG89] Nicholas Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.

[CH05] Matthew Chapman and Gernot Heiser. Implementing Transparent Shared Memory on Clusters Using Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 383–386, April 2005.

[Che88] David Cheriton. The V Distributed System. *Communication of the ACM*, 31(3), March 1988.

[Coo] Cooperative Linux. `http://www.colinux.org/`.

[DH05] John Douceur and Jon Howell. Replicated Virtual Machines. Technical Report MSR-TR-2005-119, Microsoft Research, September 2005.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[Ent] Entropia — PC Grid Computing. `http://www.entropia.com/`.

[ES03] Hideki Eiraku and Yasushi Shinjo. Running BSD Kernels as User Processes by Partial Emulation and Rewriting of Machine Instructions. In *Proceedings of the BSDCon 2003*, pages 91–102, September 2003.

[ETKY04] Toshio Endo, Kenjiro Taura, Kenji Kaneda, and Akinori Yonezawa. High Performance LU Factorization for Non-Dedicated Clusters. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '04)*, pages 678–685, April 2004.

[FD00] Graham Fagg and Jack Dongarra. FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting (PVM/MPI '00)*, pages 346–353, September 2000.

[FHH+03] Keir A Fraser, Steven M Hand, Timothy L Harris, Ian M Leslie, and Ian A Pratt. The Xenoserver Computing Infrastructure. Technical Report UCAM-CL-TR-552, University of Cambridge, January 2003.

[FK95] Bernd Freisleben and Thilo Kielmann. Automated Transformation of Sequential Divide-and-Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[FK97] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[FK98] Ian Foster and Nicholas T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (SC '98)*, November 1998.

[FK99] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[FTF+01] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10)*, pages 55–66, August 2001.

[GBD+94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, November 1994.

[GK92] David Gelernter and David Kaminsky. Supercomputing out of Recycled Garbage: Preliminary Experience with Piranha. In *Proceedings of the 6th ACM International Conference on Supercomputing (ICS '92)*, pages 417–427, July 1992.

[Gol74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, June 1974.

[GPR+98] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. GLUnix: A Global Layer Unix for a Network of Workstations. *Software Practice and Experience*, 28(9):929–961, July 1998.

[GRBK98] Edgar Gabriel, Michael Resch, Thomas Beisel, and Rainer Keller. Distributed Computing in a Heterogeneous Computing Environment. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 5th European PVM/MPI Users' Group Meeting (PVM/MPI '98)*, pages 180–187, September 1998.

[HBLL99] Mor Harchol-Balter, Tom Leighton, and Daniel Lewin. Resource Discovery in Distributed Networks. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing (PODC '99)*, pages 229–237, May 1999.

[HBS02] Hans-Jörg Höxer, Kerstin Buchacker, and Volkmar Sieh. Implementing a User-Mode Linux with Minimal Changes from Original Kernel. In *Proceedings of Linux-Kongress 2002*, pages 72–82, Semptember 2002.

[HLK03] Chao Huang, Orion Sky Lawlor, and Laxmikant V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC '03)*, pages 306–322, October 2003.

[HTC05] Yuuki Horita, Kenjiro Taura, and Takashi Chikayama. A Scalable and Efficient Self-Organizing Failure Detector for Grid Applications. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, pages 202–210, November 2005.

[Int03] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2003.

[Int05] Intel Corporation. *Intel Virtualization Technology Specification for the IA-32 Intel Architecture*, 2005.

[ITKT00] Toshiyuki Imamura, Yuichi Tsujita, Hiroshi Koide, and Hiroshi Takemiya. An Architecture of Stampi: MPI Library on a Cluster of Parallel Computers. In *Proceedings of Recent Advances in Parallel Virtual Machine and Message Passing Interface, 7th European PVM/MPI Users' Group Meeting (PVM/MPI '00)*, pages 200–207, September 2000.

[JX03a] Xuxian Jiang and Dongyan Xu. SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*, pages 174–193, June 2003.

[JX03b] Xuxian Jiang and Dongyan Xu. VIOLIN: Virtual Internetworking on OverLay INfrastructure. Department of Computer Sciences Technical Report CSD-TR-03-027, Purdue University, July 2003.

[Kan04] Kenji Kaneda. Distributed Game Tree Search on a Large Number of Idle PCs (in Japanese). In *Poster Session of the 1st Symposium on Global Dependable Information Infrastructure*, February 2004.

[KCDZ94] Peter Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 115–131, January 1994.

[KDC03] Samuel T. King, George W. Dunlap, and Peter M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 71–84, June 2003.

[KGZ+04] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. VMPlants: Providing and Managing Virtual Machine Execution Environments for Grid Computing. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (SC '04)*, page 7, November 2004.

[KHB+99] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. MagPIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. In *Proceedings of the 7th ACM SIGPLAN Symposium on Princiles and Practice of Parallel Programming (PPoPP '99)*, pages 131–140, May 1999.

[KOY05] Kenji Kaneda, Yoshihiro Oyama, and Akinori Yonezawa. A Virtual Machine Monitor for Providing a Single System Image (in Japanese). In *Proceedings of the 17th IPSJ Computer System Symposium (ComSys '05)*, pages 3–12, November 2005.

[KOY06] Kenji Kaneda, Yoshihiro Oyama, and Akinori Yonezawa. A Virtual Machine Monitor for Providing a Single System Image (in Japanese). *To appear in IPSJ Transactions on Advanced Computing Systems (ACS 13)*, 2006.

[KP02] Shay Kutten and David Peleg. Asynchronous Resource Discovery in Peer to Peer Networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS '02)*, October 2002.

[KPV01] Shay Kutten, David Peleg, and Uzi Vishkin. Deterministic Resource Discovery in Distributed Networks. In *Proceedings of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA '01)*, pages 77–83, July 2001.

[KTF03] Nicholas T. Karonis, Brian Toone, and Ian Foster. MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing (JPDC)*, 63(5):551–563, 2003.

[KTY02] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Virtual Private Grid: A Command Shell for Utilizing Hundreds of Machines Efficiently. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '02)*, pages 212–219, May 2002.

[KTY03] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Virtual Private Grid: A Command Shell for Utilizing Hundreds of Machines Efficiently. *Journal of Future Generation Computer Systems (FGCS)*, 19(4):563–573, May 2003.

[KTY04] Kenji Kaneda, Kenjiro Taura, and Akinori Yonezawa. Routing and Resource Discovery in Phoenix Grid-Enabled Message Passing Library. In *Proceedings of the 4th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid '04)*, pages 670–677, April 2004.

[LGL+96] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. `http://www.socks.nec.com/rfc/rfc1928.txt`, March 1996. Request for Comments: 1928.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[LLM88] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems (ICDCS '88)*, pages 104–111, June 1988.

[MDP+00] Dejan S. Milojíčić, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process Migration. *ACM Computing Surveys (CSUR)*, 32(3):241–299, September 2000.

[Mes] Message Passing Interface (MPI) standard. `http://www-unix.mcs.anl.gov/mpi/`.

[Mes94] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard*, May 1994.

[Mes03] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, November 2003.

[Mic] Microsoft Virtual PC. `http://www.microsoft.com/windows/virtualpc/`.

[MLV+03] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, Gaël Utard, Ramamurthy Badrinath, and Louis Rilling. Kerrighed: a Single System Image Cluster Operating System for High Performance Computing. In

Proceedings of the 9th International European Conference on Parallel Processing (Euro-Par '03), pages 1291–1294, August 2003.

[MM02] Petar Maymounkov and David Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS '02)*, pages 53–65, March 2002.

[MPI] MPICH — A Portable Implementation of MPI. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[MSK03] Masazumi Matsubara, Kazuhiro Suzuki, and Akira Katsuno. Dynamic Load Balancing in HPC applications for Autonomic Computing (in Japanese). *IPSJ Transactions on Advanced Computing Systems (ACS 3)*, 44(SIG11), 2003.

[NASa] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/Software/NPB/`.

[NASb] NAS Parallel Benchmarks in OpenMP. `http://phase.hpcc.jp/Omni/benchmarks/NPB/`.

[NLH05] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 391–394, April 2005.

[NPRC00] Michael O. Neary, Alan Phipps, Steven Richman, and Peter R. Cappello. Javelin 2.0: Java-Based Parallel Computing on the Internet. In *Proceedings of the 6th International European Conference on Parallel Processing (Euro-Par '00)*, pages 1231–1238, September 2000.

[OCD+88] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite Network Operating System. *IEEE Computer*, 21(2):23–36, February 1988.

[OOY05] Koichi Onoue, Yoshihiro Oyama, and Akinori Yonezawa. Quasar: A Mobile Computing System Based on CPU Emulator QEMU (in Japanese). In *Proceedings of the 8th JSSST SIGSYS Workshop on Systems for Programming and Applicatoins (SPA '05)*, pages 84–92, March 2005.

[Opea] OpenPBS. `http://www.openpbs.org/`.

[Opeb] OpenSSH. `http://www.openssh.com/`.

[Opec] OpenSSI (Single System Image) Clusters for Linux. `http://openssi.org/`.

[Oped] OpenSSL: The Open Source toolkit for SSL/TLS. http://www.opnessl.org/.

[OSSN02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 361–376, December 2002.

[PB94] Charles E. Perkins and Pravin Bhagwat. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '94)*, pages 234–244, August 1994.

[Pla] Platform Computin. http://www.platform.com/.

[Pov] Povray. http://www.povray.org/.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A Scalable Content-Addressable Network. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pages 161–172, 2001.

[RHWG95] Mendel Rosenblum, Stephen A. Herrod, Emmett Witchel, and Anoop Gupta. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, Winter 1995.

[RI00] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. In *Proceedings of the 9th USENIX Security Symposium*, pages 129–144, August 2000.

[RR81] Richard F. Rashid and George G. Robertson. Accent: a Communication Oriented Network Operating System Kernel. *ACM SIGOPS Operating Systems Review*, 15(5):64–75, December 1981.

[RT99] Elizabeth M. Royer and C.-K. Toh. A Review of Current Routing Protocols for Ad-hoc Mobile Wireless Networks. *IEEE Personal Communications Magazine*, pages 46–55, April 1999.

[Sar99] Luis F. G. Sarmenta. An Adaptive, Fault-tolerant Implementation of BSP for Java-based Volunteer Computing Systems. In *Proceedings of the 13th International Parallel Processing Symposium (IPPS '99) Workshop on Java for Parallel and Distributed Computing*, pages 12–16, 1999.

[SCo] SCore Cluster System Software. `http://www.pccluster.org/`.

[SD04] Ananth I. Sundararaj and Peter A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 177–190, May 2004.

[SET] SETI@home. `http://setiathome.ssl.berkeley.edu/`.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 174–184, October 1996.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*, pages 149–160, 2001.

[SN02] Gong Su and Jason Nieh. Mobile Communication with Virtual Network Address Translation. Technical Report CUCS-003-02, Department of Computer Science, Columbia University, February 2002.

[SPYH03] Osamu Sato, Richard Potter, Mitsuharu Yamamoto, and Masami Hagiya. UML Scrapbook and Realization of Snapshot Programming Environment. In *Proceedings of the 2nd International Symposium on Software Security (ISSS)*, pages 281–295, November 2003.

[Sta] Stanford Parallel Applications for Shared Memory (SPLASH). `http://www-flash.stanford.edu/apps/SPLASH/`.

[STC05] Hideo Saito, Kenjiro Taura, and Takashi Chikayama. Collective Operations for Wide-Area Message Passing Systems Using Adaptive Spanning Trees. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid '05)*, pages 40–48, November 2005.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, pages 1–14. June, 2001.

[SWE98] Charlie Scott, Paul Wolfe, and Mike Erwin. *Virtual Private Networks, 2nd Edition*. Oreilly & Associates Inc, December 1998.

[Tel01] Gerard Tel. *Introduction to Distributed Algorithms (Second Edition)*. Cambridge University Press, 2001.

[TKEY03] Kenjiro Taura, Kenji Kaneda, Toshio Endo, and Akinori Yonezawa. Phoenix: a Parallel Programming Model for Accommodating Dynamically Joining/Leaving Resources. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '03)*, pages 216–229, June 2003.

[TNS+03] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing. *Journal of Grid Computing*, 1(1):41–51, June 2003.

[TSH+99] Yoshio Tanaka, Mitsuhisa Sato, Motonori Hirano, Hidemoto Nakada, and Satoshi Sekiguchi. Resource Manager for Globus-Based Wide-Area Cluster Computing. In *Proceedings of the 1st IEEE Computer Society International Workshop on Cluster Computing (IWCC '99)*, page 237, December 1999.

[TSTS03] Hiroshi Takemiya, Kazuyuki Shudo, Yoshio Tanaka, and Satoshi Sekiguchi. Developing a Simulation System for Atmospheric Prediction on a Grid Environment (in Japanese). *IPSJ Transactions on Advanced Computing Systems (ACS 3)*, 44(SIG11):23–33, October 2003.

[TZ01] Mikkel Thorup and Uri Zwick. Compact Routing Schemes. In *Proceedings of the 13th annual ACM symposium on Parallel Algorithms and Architectures (SPAA '01)*, pages 1–10, July 2001.

[ULSD04] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. Towards Scalable Multiprocessor Virtual Machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM '04)*, pages 43–56, May 2004.

[Uni] United Devices, Inc.™. `http://www.ud.com/`.

[Use] User-mode Linux. http://user-mode-linux.sourceforge.net/.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[Vira] Virtual Iron Software. http://www.virtualiron.com/.

[Virb] Virtuozzo. http://www.swsoft.com/virtuozzo/.

[VMw] VMware Inc. http://www.vmware.com/.

[vNKB01] Rob van Nieuwpoort, Thilo Kielmann, and Henri E. Bal. Efficient Load Balancing for Wide-area Divide-and-conquer Applications. In *Proceedings of the 8th ACM SIGPLAN Symposium on Princiles and Practice of Parallel Programming (PPoPP '01)*, pages 34–43, June 2001.

[VNRS02] Jerome Verbeke, Neelakanth Nadgir, Greg Ruetsch, and IIya Sharapov. Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment. In *Proceedings of the 3rd International Workshop on Grid Computing (Grid '03)*, pages 1–12, November 2002.

[Wal02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 181–194, December 2002.

[WCG04] Andrew Whitaker, Richard S. Cox, and Steven D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 77–90, December 2004.

[WLNL03] D. Brent Weatherly, David K. Lowenthal, Mario Nakazawa, and Franklin Lowenthal. Dyn-MPI: Supporting MPI on Non Dedicated Clusters. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, page 5, November 2003.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 195–209, December 2002.

[YKaAY05] Takahiro Yamazaki, Kenji Kaneda, and Yoshihiro Oyama an Akinori Yonezawa. A Flexible and Scalable framework for Massively Multiplayer Online Games (in Japanese). In *Poster Session of Symposium on Advanced Computing Systems and Infrastructures (SACSIS '05)*, May 2005.

[YTS04] Naotaka Yamamoto, Osamu Tatebe, and Satoshi Sekiguchi. Parallel and Distributed Astronomical Data Analysis on Grid Datafarm. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing (Grid '04)*, pages 461–466, November 2004.

[ZKJ01] Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Source. Technical Report UCB/CSD-01-1141, University of California at Berkeley, April 2001.