On Managing Execution Environments for Utility Computing

Roy S.C. Ho, David C.M. Lee, Daniel H.F. Hung, Cho-Li Wang, and Francis C.M. Lau

Department of Computer Science
The University of Hong Kong, Hong Kong

Email: {*scho,cmlee,hfhung,clwang,fcmlau*}*@cs.hku.hk*

ABSTRACT

The main goal of utility computing is to offer computing resources on-demand. With this paradigm, applications could be dispatched to remote platforms for execution. A fundamental issue is whether the *execution environments* (*EEs*) in remote platforms fit the requirements of the user applications. Current approaches such as service-oriented grid middleware may not always be able to provide the EEs that best meet the requirements of various applications. In this paper, we (1) propose a software framework for managing EEs over a network; and (2) present a reference implementation of the framework, called SLIM, for constructing Linux-based EEs. In contrast to the traditional network booting method, SLIM opts for optimal file sharing for individual directories according to their access patterns, which shortens the construction time of EEs and improves their run-time performance. Besides, an in-memory execution mode is supported which establishes customized EEs in compute nodes without affecting the OS/data stored in the permanent storage. Experimental results demonstrate the efficiency of SLIM in a network involving 272 machines.

Categories and Subject Descriptors

C.5.5 [**Computer System Implementation**]: Servers; D.4.m [**Operating Systems**]: Miscellaneous

General Terms

Management, Design, Performance, Experimentation

Keywords

Utility computing, execution environments, system administration, network booting, grid computing

1. INTRODUCTION

Advances in commodity networking and computing technologies have made possible the sharing of a wealth of computing resources among users regardless of their physical locations. The development of grid middleware such as the Globus Toolkit [19] represents a major step forward in that possibility by introducing standard protocols and semantics for resource access. The current trend is towards computing resources as *utility*, where computing resources can be offered or obtained *on-demand*. The *utility computing* paradigm features cost-effectiveness, user's convenience, and extensibility of user applications and computing platforms.

While current R&D efforts have been focusing on how to aggregate (e.g., [21][15][4][23][24]) and make use of (e.g., [19][20][1]) distributed computing resources, relatively little attention is paid to the fact that utility computing essentially *decouples* the computing logics (i.e., the applications) from the computing platforms. This decoupling introduces a potential mismatch between the configurations of the platforms and the requirements of the applications, which could lead to application failures (incompatible binaries), suboptimal performance (lack of native supports for performance optimization), and users' inconveniences (they need to learn the usage of the new environments and re-configure/engineer their applications). Therefore, an important issue with utility computing is how to provide the customized *execution environments* (*EEs*) for different applications.

Traditionally, multiple EEs are installed in different disk partitions in a computer. This approach to managing EEs is costly. The resultant cost is a compound of: the number of machines to be managed, the number of EEs, and the complexity of management tasks associating with each EE (e.g., OS/software installation, configuration, and update; data backup, etc.). Another approach to EE management is network booting (e.g., [2]). It generally requires the entire root directory to be mounted through a network file system, which often results in poor run-time performance as some files are frequently updated/accessed; retrieving them through the network is inefficient.

We address the problems of managing EEs for utility computing. Specifically, we propose a framework for a network service to be used for managing Unix-like EEs, and constructing them in distributed platforms. The target distributed platforms include the tightly-coupled ones (clusters), the loosely-connected (home/desktop computers), and the hybrid ("commodity grids" consisting of distant clusters). We also present a reference implementation of the framework, called *SLIM* (Single Linux Image Management) [7], which can efficiently disseminate the Linux kernel to distributed computers upon system boot-ups. Customized EEs which include shared libraries and user data are made available to the remote platforms via *discriminative* file sharing mechanisms. We evaluate the performance of SLIM on a network involving 272 PCs. In addition, we provide preliminary results of using SLIM on broadband networks, which indicate the feasibility of constructing utility computing platforms in a pervasive fashion.

The rest of this paper is organized as follows. Section 2 outlines the related work. Section 3 describes the design objectives and system architecture of the software framework. Section 4 presents SLIM. Case studies and some experimental results are given in Sections 5 and 6, respectively. We conclude the paper in Section 7.

2. RELATED WORK

There is a large body of literature related to resource sharing in distributed computers. We classify them into four main categories: grid middleware, job schedulers, application-specific solutions, and run-time support.

Grid computing has become a popular research topic in recent years. Most research efforts have been focusing on grid middleware; examples include the Globus Toolkit [19], Legion [21], Condor-G [20], and UNICORE [15]. While these middleware provide convenient mechanisms for resource sharing and the needed security support, they demand the applications to be either developed with a special library, or executed in application/service containers. Therefore, this approach does not seem to be ideal for most legacy applications due to the required porting effort. Furthermore, some service-oriented grid middleware such as the Globus Toolkit might not be suitable for traditional HPC applications due to the completely different programming paradigms. It is indeed desirable to allow an application to harness remote computing resources without any modification.

Job schedulers (e.g., [4], [23], [24], etc.) have been a widely adopted approach to aggregating and sharing distributed resources. Although a job scheduler can dispatch an application to a platform that supports the desired EE, the type of applications which can utilize the platform is still confined by the existing EEs installed in the platforms; supporting new types of applications would require tedious system administration (e.g., OS/software re-installation, configuration, etc.).

There have also been a number of application-specific solutions proposed by the industry such as the the Data Center Markup Language (DCML) [1]. These solutions meet well the needs of specific forms of resource sharing (e.g., access to databases, etc.); they however do not provide the needed flexibility for supporting the existing general applications. Run-time supports such as service provisioning [22][18], run-time adaption [17], and stub generation (e.g., [16]) fail to provide a transparent support; modification of existing applications is therefore needed.

To conclude, current research appears to fall short of efficiently providing a most suitable EE that an application *intends* to run on. The result is user's inconvenience which in turns limits the adoption. In our research, we aim to provide a better solution to this problem.

3. SOFTWARE FRAMEWORK

In this section, we present the design objectives and system architecture of an EE management service whose function is to disseminate the needed EEs to networked machines and to construct them on-demand, according to the requirements of the target applications. The goal is to facilitate resource sharing in utility computing environments by bridging the gap between the application demands and the configuration of the computing platforms.

3.1 Design Objectives

The framework of the EE management service aims to fulfill the following objectives.

Convenient system administration. Common management tasks associating with a particular EE include installation, configuration, software upgrade/update, and backup. If these tasks have to be repeated for multiple EEs in a large number of machines (which might even be geographically distributed, such as in a grid), the induced cost and effort could well offset the benefit of sharing the computing resources. Therefore, EEs should be centrally managed; the configurations required in the individual compute nodes should be performed only once and kept simple.

Efficient EE construction. A running machine should only need to reboot in order to obtain a specific EE from the service. Furthermore, the EEs should be quickly constructed in order to shorten the delay in switching from one EE to another. To this end, the framework should accommodate different implementation strategies for performance optimization in various network/system configurations.

Complete transparency. The network service should be transparent to the users and applications, i.e., no modification of the applications is needed. Furthermore, users should not notice the *nature* of a platform. For example, they should be able to utilize an ad-hoc computing platform consisting of idle computers as if they were using dedicated cluster computers.

Platform/network neutrality. The design should aim at a generic service which imposes no restriction on the platforms and network interconnects; and is able to cater to different types of platform ranging from dedicated systems serving compute-intensive jobs to home computers having idle cycles to share.

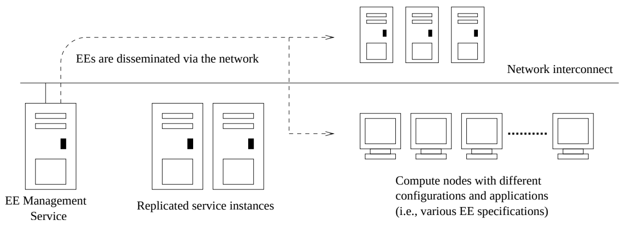
EEs are disseminated via the network
Network interconnect
EE Management Service
Replicated service instances
Compute nodes with different configurations and applications (i.e., various EE specifications)

Figure 1: EE Management Service and Compute Nodes

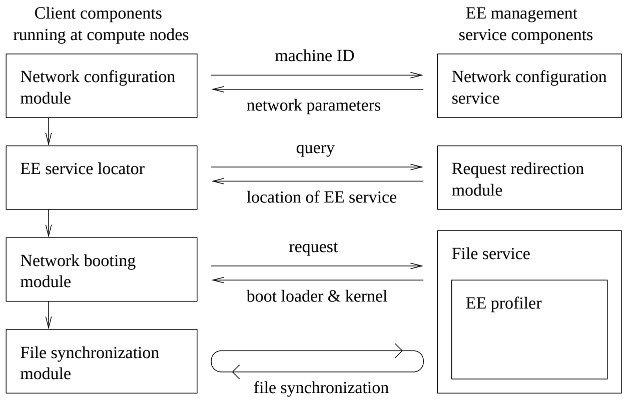
Client components running at compute nodes
EE management service components
Network configuration module
machine ID
network parameters
Network configuration service
EE service locator
query
location of EE service
Request redirection module
Network booting module
request
boot loader & kernel
File service
EE profiler
File synchronization module
file synchronization

Figure 2: Client and Server Components

In-memory execution. Most existing compute nodes are installed with a primary OS. Our framework intends to support an optional in-memory execution mode which establishes the EEs entirely in the physical memory. This feature facilitates access to idle computers as resources for utility computing.

3.2 System Architecture

In our framework, an EE is essentially a ready-to-run OS image including the shared libraries/utilities needed by end users' applications. The client/server paradigm is adopted in designing the EE management service, which is depicted in Figure 1. As shown in the figure, all EEs are stored in and managed by an EE management server, while the compute nodes would obtain the EEs from the server on-demand.

The framework comprises several client and server components. The client components executing in the compute nodes include (1) a network configuration module, (2) an EE service locater, (3) a network booting module, and (4) a file synchronization module. The server components, which constitute the EE management service, include (1) a network configuration service, (2) a request redirection module, (3) a file service, and (4) an EE profiler.

Figure 2 illustrates the interactions between these components. When a compute node boots up, the network configuration module sends or broadcasts a machine ID (which uniquely identifies that machine) to a network configuration service for obtaining the network configuration parameters such as an IP address. Then, the EE service locater sends a query to the request redirection module to find out the location of the EE dissemination service. Based on the originating address of the query (i.e., the identity and location of the compute node) and the current load of each replicated dissemination service, the request redirection module returns the network address of the best available dissemination service to the compute node.

At this point, the compute node starts to retrieve the EE from the file service. The file service relies on the EE profiler to identify the specification of each EE to be disseminated. The profiler keeps track of (1) a list of EEs (and a list of directories for each EE) being managed; (2) the file synchronization specification, which specifies *how* the individual directories are disseminated to the compute nodes (e.g., through NFS, or copy directly); (3) a list of hardware profiles and drivers for different compute nodes; and (4) a table which maps the nodes' IP addresses to the corresponding hardware profiles and the default EEs to be disseminated. These lists and tables are stored in the machine that hosts the file service. They are prepared by the system administrators, and are retrieved when a compute node requests an EE from the EE management service.

The dissemination process begins at this point: the network booting module at the client side obtains a boot loader from the file service, which is for the user to opt for booting from the local OS'es or from one of the kernels provided through the network. It should be noted that the boot loader is in fact optional and is only used for directly interacting with users (i.e., not for compute servers). If the machine is configured to skip the boot loader or the user opts to boot from network, the network booting module would retrieve the default or chosen kernel from the file service and boot with it.

When the booting process finishes, the file synchronization module will obtain the file synchronization specification, which specifies how the individual directories of the EE are made available from the central storage server(s) to the compute node. For example, files that are subject to modifications should be copied locally for better performance, while those files that are rarely retrieved could be shared via a network file system. The construction of the EE would finish after the file synchronization module performs the needed file transfers and establishes the connections to the network-shared volumes.

4. SLIM: SINGLE LINUX IMAGE MANAGEMENT

We present a reference implementation of the EE management service framework called SLIM, a network service to be used for managing and disseminating Linux-based EEs to distributed PC systems. SLIM offers the following unique features.

- Manual installation of the Linux OS in individual PCs is completely avoided. SLIM instantly turns a PC into a Linux workstation through the network. It

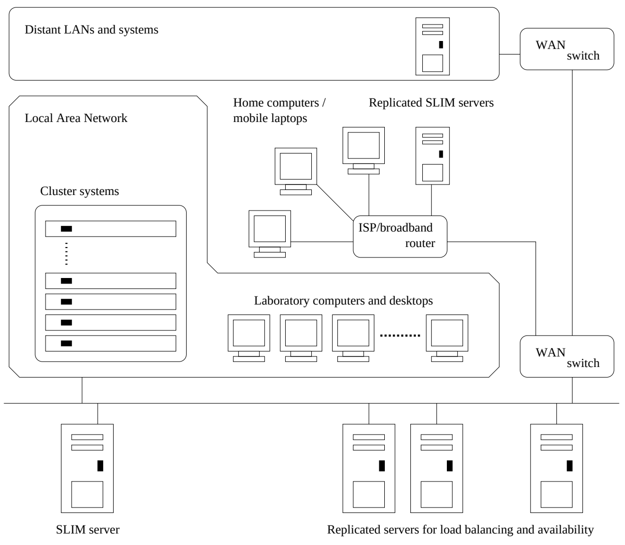

Figure 3: Typical SLIM Deployment Scenarios

does not affect or depend on any OS'es pre-installed in the hard disk

- Fast system recovery and backup. Compute nodes do not hold important user data; system administrators only need to perform backup at the central SLIM servers.
- SLIM does not impose any restriction on the Linux OS, which could be tailored to meet the requirements of different applications and deployment scenarios. It does not require any modification of the user applications, either.

Figure 3 illustrates the typical deployment scenarios of SLIM. As shown in the figure, SLIM is designed to serve both the machines residing in the same LAN and those that are connecting to SLIM via the Internet and broadband networks. In essence, all computing resources on these networks can be utilized on-demand with the EEs being managed and distributed by the SLIM service.

4.1 System Overview

SLIM leverages legacy firmware/software supports for performing network configuration at and disseminating the EEs to remote compute nodes. These supports include the pre-boot execution environment (PXE) [5], dynamic host configuration protocol (DHCP) [10], trivial file transfer protocol (TFTP) [6], `rsync` [14], and network file system (NFS) [3]. Apart from all these, all software modules are merely a collection of *shell scripts*, which indicates the simplicity and portability of the EE management framework and the SLIM prototype. The implementation strategies of SLIM for LAN are summarized in Table 1.

There are three system processes running at the server side which collectively provide the SLIM service: the DHCP server sends the network configuration parameters (e.g., IP addresses, etc.) and the IP address of the TFTP server to the compute nodes; the TFTP server delivers the Linux kernel and a custom initial ramdisk (`initrd`) to the nodes; the NFS server hosts the pre-installed Linux system images for different EEs. While these processes could execute in a single server machine in a small deployment, they are actually independent of each others and therefore could be distributed (and/or replicated) in multiple machines for higher performance and scalability.

Table 1: Implementation Strategies of SLIM on LAN

Operation	Strategy
Network configuration	DHCP
Locating EE service	location given by DHCP
Network booting	PXE/TFTP
File synchronization	`rsync`/NFS

We aim at minimizing the management effort on the compute node side. In SLIM, this is achieved by employing PXE as the enabling mechanism for network booting. System administrators will need only to (1) enable the PXE feature in the basic input/output system (BIOS) of a PC, (2) connect the PC to the physical network which hosts the SLIM service, and (3) turn on the power to start booting with PXE. When an administrator wishes to launch another EE, he needs only to map the compute node's IP address to a different EE in the SLIM server, reboot the node and repeat the booting process.

Figure 4 gives an overview on the process of constructing an EE in a compute node, which is discussed in details in the following sections.

4.2 Network Configuration and Booting

When a PC in a SLIM network is turned on, it would first perform the power on self test and initialize the hardware. After that, the PXE routine stored in the firmware will obtain an IP address and a bootstrap program through the network. This operates as follows. First, the PXE module broadcasts a `DHCP Discover` message over the network. If that node is found eligible to obtain an EE, the DHCP server of the SLIM service would reply to the node with a `DHCP Offer` message, which includes the IP address being "leased" to that node, and the IP address of the SLIM TFTP server. Upon receipt of the IP addresses, the node would connect to the TFTP server to retrieve the network bootstrap program, which is called `pxelinux`. The purpose of `pxelinux` is to download the Linux kernel and the `initrd` from the TFTP server. The `initrd` is basically an image of a "mini-root" file system, which is compressed as an archive for efficient transfer. `pxelinux`, after receiving the kernel and `initrd`, would uncompress `initrd` and load the image together with the kernel into the physical memory, and then start the booting process which performs the following tasks.

1. Initialize the hardware and load the corresponding driver modules according to the hardware profile of

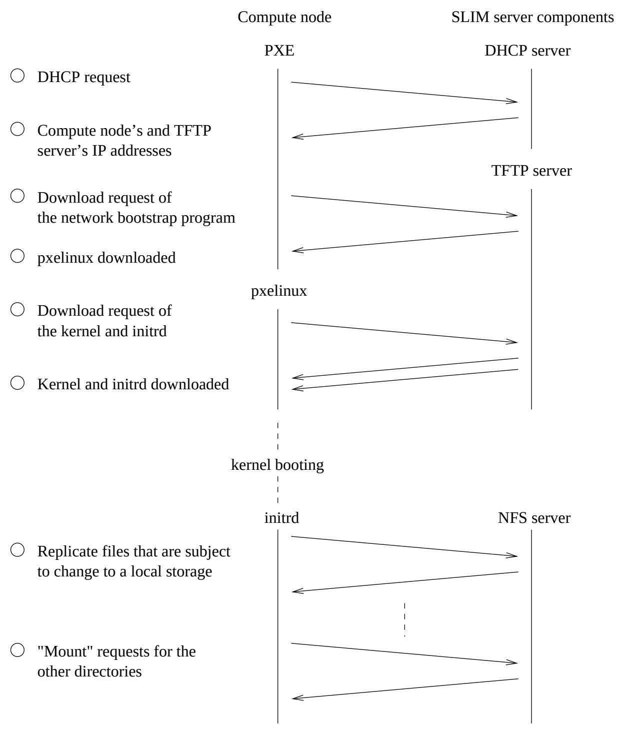
Compute node
SLIM server components
PXE
DHCP server
DHCP request
Compute node's and TFTP server's IP addresses
TFTP server
Download request of the network bootstrap program
pxelinux downloaded
pxelinux
Download request of the kernel and initrd
Kernel and initrd downloaded
kernel booting
initrd
NFS server
Replicate files that are subject to change to a local storage
"Mount" requests for the other directories

Figure 4: Process of EE Construction

that compute node

2. Mount the root directory of the disseminated EE from the NFS server for file synchronization
3. Copy files to a local storage and mount the shared NFS volumes according to the file synchronization specification (more on this in the next section)
4. Switch to the file system on the local storage as the root file system
5. Start the `init` program to carry out normal system initialization process

4.3 File Synchronization

The file synchronization process is performed in two stages: (1) copy the system files that are subject to modifications to a local storage, and (2) mount the other directories through NFS. The rationale of these arrangements is as below.

In general, there are two ways to retrieve the OS image from a central file server. First is to copy the files from the server to local storage; second to access them through a network file system such as the NFS. Deciding which approach is better is nontrivial, which is in fact a trade-off between the booting time and the run-time performance: it takes time to copy a file during the booting process, but once a file is stored locally, subsequent accesses (especially updates) will be faster. In order to achieve a good balance, we have experimented with various possibilities in synchronizing files, and have come up with the following two experiences.

NFS optimization. We found that the configuration options of the NFS affect I/O performance substantially. This phenomenon actually stems from the fact that most files are read-only in an OS image. In a newly installed Fedora Core 1 image [11], for example, 86% of files and 96.3% of bytes are read-only during run-time (Table 2). Nevertheless, the default configuration options of NFS target at read/write access which is inefficient. The maintenance of file and meta-data consistency induces much unnecessary network traffic and thus decreases performance. In order to address this issue, we set the following options in the NFS configuration file for each shared EE: "`ro`", which specifies a shared directory as read-only; "`nolock`", which disables file locking; and "`noatime`", which avoids updating the file access time stamps. In addition, the "`actimeo`" value is greatly increased, which determines the life time of files being cached in an NFS clients. We found that these changes do improve performance considerably.

Discriminative file sharing mechanisms. We observed that the booting time increases substantially with the number of files to be copied. This is caused by the round-trip latencies during file copies and the network congestion at the central NFS server. Therefore, copying most files of an entire OS image (whose size is in the order of GB for modern Linux distributions) is impractical. However, frequently updating files through NFS would affect run-time performance. In order to tackle these issues, SLIM adopts a *discriminative* approach to file sharing, which allows administrators to select the optimal file sharing mechanisms according to the access pattern of individual directories. For example, the default file sharing strategy in the current SLIM prototype leaves all read-only directories in the NFS server. Besides, all user data (i.e., their home directories) are also shared via NFS so that users can access their data from any machines. To avoid file updates through the NFS, SLIM copies all system files which are subject to modifications to the local storage of compute nodes. These strategies could be customized according to the specific needs of applications. We believe these arrangements do achieve a good trade-off between the conflicting goals of minimizing booting time and optimizing the run-time performance. In fact, a later version of SLIM has been managing several HPC clusters since 2002 and most users have not noticed significant performance degrade caused by the NFS activity.

4.4 Options for File Copy

The files copied from the SLIM server have to be stored locally. SLIM supports three options for file copy: (1) full-copy to RAM; (2) full-copy to hard disk; and (3) copy-if-needed.

Full-copy to RAM. As shown in Table 2, the amount of files to be copied (i.e., the "read/write" ones) is relatively small (around tens of megabytes). Considering that modern PC systems are often equipped with 512 MB or even gigabytes of physical memory, it is feasible to copy all

Table 2: Access Pattern of the Fedora Core 1 Image

Directory	Read/write	Number of files (%)	Size in MB (%)
`/bin`	read-only	98 (0.0)	4.6 (0.2)
`/boot`	read-only	37 (0.0)	4.6 (0.2)
`/dev`	read/write	18704 (10.8)	0.4 (0.0)
`/etc`	read/write	4122 (2.4)	25.0 (1)
`/lib`	read-only	3920 (2.3)	76.9 (3.2)
`/sbin`	read-only	274 (0.2))	12.2 (0.5)
`/usr`	read-only	144202 (83.5)	2196.3 (92.1)
`/var`	read/write	1410 (0.8)	65.0 (2.7)

these files to the memory instead of the hard disk. SLIM supports this feature by creating a *ramdisk* in the physical memory. A complete *in-memory execution* has two advantages. First, it avoids the need to plan ahead for the hard disk usage, which is especially desirable in utility computing environments where a machine might execute different EEs over time. Second, it preserves the OS/user data originally stored in the permanent storage of a compute node.

Full-copy to hard disk. Files can be stored in hard disks if the memory space is scarce. In this case, a spare hard disk partition is assumed to be in place for file storage.

Copy-if-needed. This is a variant of the previous option in which the files to be copied (or some of them) have already been stored in the hard disk. In this case, SLIM would compare the files stored in the compute node with that in the central server, and would only copy those that are new or have been updated. For the updated files, SLIM would only transfer the differences (i.e., the "diff") between the old and the new versions so as to speed up the dissemination process. This option offers better performance than the others as network communication would mainly involve version checking instead of the whole set of files. In SLIM, these features are implemented by using the `rsync` protocol.

4.5 Optimization Techniques for WAN

Although SLIM has been evolved in LAN environments, we aim to extend its service for the machines scattered over the Internet. Specifically, we adopt two techniques for optimizing the performance in disseminating EEs through a WAN. First, the NFS server is tuned to operate with the TCP protocol instead of UDP in order to tackle the frequent packet loss in WAN. This is because the *exponential backoff* for TCP retransmission handles packet loss efficiently, which makes TCP a better choice over UDP. Besides, a packet loss in UDP implies the whole NFS/RPC request has to be retransmitted. TCP, by contrast, only needs to retransmit that lost packet.

Second, in order to minimize the long latency incurred during file transfers and version checking, all files that need to be copied are packed into a compressed archive file. This strategy effectively decreases the number of file queries and thus the booting time.

5. CASE STUDIES

In this section, we describe how SLIM is deployed in our department. We also outline some interesting applications that might benefit from a network service like SLIM.

5.1 SLIM in HKU CS

SLIM has been managing the EEs for 600+ machines since 2002 in the Department of Computer Science, HKU. Like other CS departments in most academic institutions, there is a wide range of machines in our department. Most of them are ordinary PC systems, which are mainly used for three types of applications: (1) HPC clusters, (2) teaching, and (3) students' desktops.

HPC clusters. We have four PC clusters (350+ nodes) having different hardware configurations, which range from the Pentium II type to Pentium IV, and a few AMD symmetric multiprocessors (SMPs). We found that managing the EEs for these computers for different users could be nontrivial. Specifically, some systems research projects demand different versions of Linux/GNU libraries/JVMs (Java Virtual Machines); our collaborators in the Asia-Pacific Grid [8], China National Grid [9], and Hong Kong Grid [13] occasionally request to run experiments and demos in our cluster nodes—each of these scenarios might require a specific EE. In the past, we divided the cluster nodes into multiple partitions in order to satisfy the different needs, but the main problems are the poor utilization of some partitions and the difficulty in resizing the partitions to satisfy the changing demands (resizing usually implies re-installation/customization of EEs). SLIM solves the problem very well as any EE can be disseminated on-demand according to the users' needs and the hardware specifications of the target platforms.

Teaching. The computers used for teaching encounter a similar problem. For instance, the requirement of the OS course differs substantially from that of the real-time systems course. In these cases, SLIM efficiently constructs the needed EEs right before the laboratory sessions according to the requirements given by the instructors.

Students' desktops. These computers are generally installed with the Windows OS. However, some students might need to use the Linux OS as a programming environment or "X-terminal" to our Solaris servers. In these cases, SLIM could deliver the EEs which execute entirely in the physical memory. Furthermore, since most of these computers are idle at nights, they could be restarted with the Linux OS for HPC jobs submitted by the users in our department and the other faculties.

5.2 Potential Applications

We outline some potential applications of SLIM. First, home computers have a very good potential in forming powerful platforms for massive computations as they generally have a lot of idle resources. SLIM could facilitate this computing paradigm by disseminating the needed EEs over the broadband networks. Advantages aside, however, disseminating EEs over the Internet imposes new challenges as the performance of typical residential broadband networks is much lower than that of the LAN. We have started to address this issue and obtained some preliminary results, which are presented in the next section.

Second, SLIM could potentially support utility computing services for the society. For instance, many small/medium enterprises (SMEs) would like to adopt the Linux OS and the open-source software for cost-effectiveness. However, most of them do not have in-house technical support for Linux. In this connection, the utility computing service model finds its place by centralizing the software maintenance and/or providing computing resources on-demand. The service could possibly be offered by the Internet service providers (ISPs) since they are "physically closer" to the end users on the network.

6. EXPERIMENTS

In this section, we evaluate the performance of the SLIM prototype. Section 6.1 presents the experimental results on LAN. Section 6.2 reports some preliminary results on the Internet.

6.1 SLIM on LAN

We conducted several experiments with SLIM in the HKU CS Gideon cluster [12]. The cluster consists of 300 Pentium IV machines and two Ethernet networks; each machine is equipped with a 40GB IDE hard disk and 512 MB RAM. The first Ethernet network is for EE management while the second for high-speed inter-process communication. The management network is organized in a hierarchical manner: 13 24-port Fast Ethernet switches (each interconnects 22-24 nodes) are interconnected by a Gigabit Ethernet switch. In our experiments, all server processes of the SLIM service were hosted by a single Pentium IV machine with 512MB RAM and an IDE hard disk. The SLIM service disseminated the Fedora Core 1 OS to the cluster nodes by sharing the `/bin`, `/lib`, `/sbin`, and `/usr` directories through the NFS; while copying the others (e.g., `/etc`, `/dev`, etc.) to the nodes' local storage. We evaluated the performance of SLIM under three file copy options: *full-copy to RAM*, *full-copy to HD*, and *copy-if-needed*. For the last option, the ideal case was tested where all files had been stored locally in the compute nodes and none of them needed to be updated. The results are shown in Figure 5.

As shown in the figure, *copy-if-needed* offers the best re-

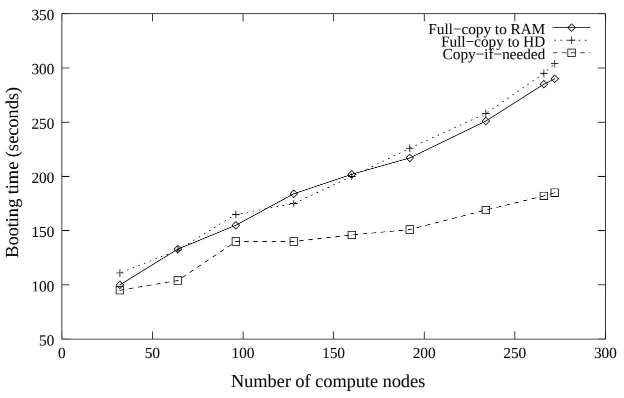

Figure 5: Booting Time vs. Number of Compute Nodes

sults as the dissemination process only involves the transfers of the kernel and the queries for version checking. The other two options offer a similar performance, which suggests the performance bottleneck lies more on the throughput of the SLIM server than on the hard disk I/O at the compute nodes. Nevertheless, the performance bottleneck can be relieved by replicating the SLIM service (particularly the NFS server) to multiple machines.

Overall, the experiments demonstrate that an EE can be constructed in 3 ("copy-if-needed") to 5 ("full-copy") minutes in 272 PC systems, which proves that SLIM is indeed efficient in constructing Linux-based EEs.

6.2 SLIM on the Internet

In this test, the same Fedora OS image is disseminated to a compute node which was connected to our departmental SLIM service via a broadband link with a download bandwidth of 3 Mbps. Compared to the LAN experiment, this test involved two additional optimizing techniques: using TCP as the transport protocol and packing the files in an archive file to avoid round-trip latency in copying the files individually. The EE (for only one compute node) took 3 minutes 40 seconds to construct. Although the performance is considerably lower than that in the LAN, the booting time still falls within a reasonable time frame. Consider the very limited bandwidth and long network latency in the testing environment, it is indeed an encouraging result that motivates us to perform further optimization.

7. CONCLUDING REMARKS

We propose a software framework for a network service to be used for managing EEs, and constructing these EEs in distributed computers for on-demand utility computing. We also present SLIM, a reference implementation of the framework for efficient dissemination of Linux-based EEs. The experimental results show that SLIM is able to efficiently construct the EEs in 272 machines in 3 ("copy-if-needed") to 5 ("full-copy") minutes. The preliminary result of the WAN experiment suggests that using SLIM on the Internet is feasible, but needs further investigation on possible performance improvements.

Future work will focus on performance optimization of SLIM in WAN, which will be conducted in three dimensions. First, we will develop a custom file system to mask the server/network failures and the long network latency in WAN. In this file system, *partial file serving* will be implemented which allows the usage of a file before it is completely fetched from the SLIM server; which should improve the responsiveness of applications. Second, we aim to design some negotiation protocols to be used by a compute node to negotiate with the SLIM service for the optimal size and content of an EE based on the hardware capability of the compute node, the real-time network performance, etc. Third, we plan to incorporate more "intelligence" in the discriminative file sharing policies, by taking into account the past usage history, users' preferences, the intended usage of a compute node, etc., to determine the optimal file sharing strategy.

8. ACKNOWLEDGEMENT

This research is supported in part by the HKU Foundation Seed Grant 28506002 and the HKU Large Equipment Grant 01021001.

9. REFERENCES

[1] Data Center Markup Language. http://www.dcml.org/.

[2] Network Boot and Exotic Root HOWTO. http://www.linuxforum.com/linux-network-boot.php.

[3] NFS on Linux. http://nfs.sourceforge.net/.

[4] Portable Batch System. http://www.openpbs.org/.

[5] Preboot Execution Environment (PXE). `ftp://download.intel.com/labs/manage/wfm/download/pxespec.pdf`.

[6] RFC 1350 – The TFTP Protocol. http://www.faqs.org/rfcs/rfc1350.html.

[7] SLIM: Single Linux Image Management. http://slim.csis.hku.hk/.

[8] The Asia-Pacific Grid. http://www.apgrid.org.

[9] The China National Grid. http://www.chinagridforum.org/.

[10] The Dynamic Host Configuration Protocol (DHCP). http://www.dhcp.org/.

[11] The Fedora Core 1 Linux OS. http://fedora.redhat.com/.

[12] The HKU CS Gideon Cluster. http://www.srg.csis.hku.hk/gideon/.

[13] The Hong Kong Grid. http://www.hkgrid.org.

[14] The rsync Utility. http://samba.anu.edu.au/rsync/.

[15] The UNICORE Forum. http://www.unicore.org/.

[16] J.R. Callahan and J.M. Purtilo. A Packaging System for Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, 17(6):626–635, June 1991.

[17] F. Chang and V. Karamcheti. Automatic Configuration and Run-Time Adaptation of Distributed Applications. In *Proc. of the 9th International Symposium on High-Performance Distributed Computing*, pages 11–20, August 2000.

[18] T. Dimitrakos et al. An Emerging Architecture Enabling Grid Based Application Service Provision. In *Proc. of the 7th IEEE International Enterprise Distributed Object Computing Conference*.

[19] I. Foster, C. Kesselman, J.M. Nick, and S. Tueckel. The Physiology of the Grid - An Open Grid Services Architecture for Distributed Systems Integration. In *White Paper, The Globus Project. http://www.globus.org/.*

[20] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5:237–246, 2002.

[21] A. Grimshaw, A. Ferrari, A. Knabe, and M. Humphrey. Legion: An Operating System for Wide-Area Computing. *IEEE Computing*, 32(5):29–37, May 1999.

[22] X. Jiang and D. Xu. SODA: A Service-On-Demand Architecture for Application Service Hosting Utility Platforms. In *Proc. of the 12th IEEE International Symposium on High Performance Distributed Computing, 2003*, pages 174–183, June 2003.

[23] T. Tannenbaum, D. Wright, K. Miller, and M. Livny. Condor – A Distributed Job Scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[24] S. Zhou. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Workshop on Cluster Computing*, 1992.