InstantGrid: A Framework for On-Demand Grid Point Construction

Roy S.C. Ho, K.K. Yin, 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,kkyin,cmlee,hfhung,clwang,fcmlau}@cs.hku.hk

Abstract

This paper proposes the InstantGrid framework for on-demand construction of grid points. The framework comprises the following components: (1) a centralized model of software management using an application-centric software grouping scheme; (2) proactive configuration of grid middleware, which shortens the time in composing and switching between execution environments; (3) performance optimization techniques using I/O caching and discriminative file sharing mechanisms; and (4) an in-memory execution mode that enables a machine to participate in grid without affecting the OS/data stored in the permanent storage. Compared with traditional approaches, this new framework is designed to substantially simplify software management in grid systems, and is capable to instantly turn any computer (be it a cluster node or a desktop PC) into a grid-ready platform with the desired execution environment. The advanced features also facilitate ad-hoc formation of grid platforms in computers having idle resources. We describe a reference implementation of InstantGrid for constructing Linux-based grid points. Experimental results demonstrate that a 256-node grid point with commodity grid middleware can be constructed in 5 minutes from scratch.

1 Introduction

Grid has become a viable approach to building scalable platforms for on-demand utility computing. To construct a *grid point*, system administrators generally have to install and configure a frontend machine acting as the gatekeeper; and an OS, a client module for scheduling, and other application libraries in each compute node. While these tasks seem to be manageable, constructing grid points in this manner actually limits the power and flexibility of grid computing in the following three aspects.

First, grid computing *decouples* the computing logics (i.e., the applications) from the computing platforms as any computing resource could be consumed by an arbitrary foreign application, which is in contrast to the traditional case where the intended applications are usually well-defined. This phenomenon introduces the potential mismatch between the configurations of the platforms and the requirements of the applications. Mismatches might occur when an application is dispatched to different computing platforms with various configurations; or when a platform such as a cluster is being shared by multiple applications with deviating requirements. Therefore, grid systems demand a custom model for managing multiple execution environments (EE's) in order to provide flexible application support. Existing approaches to constructing and managing grid points, however, seem to be lacking in this regard.

Second, contemporary OS'es and middleware (e.g., libraries for message-passing, job schedulers, gatekeeper systems, etc.) are assumed to be installed and configured *separately* and only *once*. This is acceptable in traditional distributed computing since the target applications (and hence the EE's) are known clearly and do not change often. In grid computing, however, supporting EE's for different applications might result in frequent re-installation and re-configuration of the

OS'es and middleware, which significantly complicates system administration. Furthermore, it could take long to switch between EE's, which results in poor users' experience and system utilization. Therefore, it is desirable to have efficient mechanisms to construct an EE, and to switch from one EE to another.

Finally, the OS'es, middleware, and user data in existing computing platforms are normally stored in the permanent storage. Consequently, people are reluctant to share computational resources to foreign applications since this might require different OS and systems software support; wide adoption of the grid computing paradigm is therefore prohibited.

While the current R&D efforts have been focusing on how to aggregate (e.g., [13][12][3][20][21]) and make use of (e.g., [17][18][1]) distributed computing resources, few of them have addressed the above issues.

1.1 Our Solution

We propose *InstantGrid*, a framework for efficient construction of grid point. This new framework is designed to simplify software management in grid systems, and is capable to instantly turn any computer into a grid-ready platform with optimized runtime performance. The EE's are centrally managed in an InstantGrid server, and can be disseminated to and launched in remote compute nodes upon system boot-ups. The advanced features also facilitate ad-hoc formation of grid platforms in idle computers. The framework comprises the following core components.

Centralized and application-centric software management. All OS images and grid middleware are stored and managed in a central InstantGrid server. These software components are grouped into distinct, pre-defined, EE's; each EE targets at a specific type of applications. For example, service-oriented distributed applications and job submission-based HPC rely on two very different EE's. This model guarantees well-defined EE's for (and hence compatibility with) various grid applications. The centrally managed EE's are *disseminated* to the compute nodes on-demand through the network, according to the application requirements.

Proactive software configuration. Instead of installing and configuring OS'es and middleware incrementally after they are disseminated to the compute nodes, all software components in a specific EE are required to be pre-configured in the InstantGrid server. In other words, software would not be disseminated to the compute nodes unless all of them are ready to be executed to form the desired EE. These approaches shorten the time in composing and switching between EE's.

Performance optimization techniques. The centralized management model implies an entire EE (which could be as large as a few gigabytes) has to be disseminated to the compute nodes on-demand. While replicating all files is obviously impractical, the existing network booting approaches which completely rely on the network file system (NFS) would result in poor runtime performance. We aim to address this problem by exploiting efficient I/O caching techniques to avoid excessive file transfer. In addition, the *discriminative file sharing mechanisms* select the suitable strategy (e.g., NFS-shared, replication, etc.) according to the usage pattern of a file, which optimizes both the dissemination and runtime performance.

In-memory execution mode. We aim to cater for a scenario in which the data/OS stored in the permanent storage in the compute nodes would not be altered (or even accessed) when an EE obtained via the network executes, i.e., a complete in-memory operation. This is especially useful for supporting grid computing in existing cluster platforms, desktop/home computers, and diskless blade servers.

We developed a reference implementation of InstantGrid, which manages and disseminates the commodity Linux OS and grid middleware to construct production grid environments. In our design, the performance optimization techniques and the in-memory execution support are integrated into a toolkit called *SLIM* (Single Linux Image Management), which forms the low-level support for disseminating EE's in InstantGrid. While InstantGrid is specifically designed for grid point construction, the SLIM component could be used for convenient software management and system administration in distributed systems. We conducted experiments with the implemented prototype, the results demonstrate that a 256-node grid point can be constructed in 5 minutes from scratch. This grid point was equipped with the Fedora Linux Core 1, Globus Toolkit 3, Portable Batch System (PBS), and the Ganglia cluster monitoring package.

The rest of this paper is organized as follow. Section 2 outlines the related work. Section 3 describes the design objectives and system architecture of InstantGrid. Section 4 presents SLIM. Some case studies and experimental results are given in Sections 5 and 6, respectively. We conclude this research in Section 7.

2 Related Work

There have been much literature related to building platforms for remote resource sharing. We classify these related work 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 designing grid middleware, examples include the Globus Toolkit [17], Legion [13], Condor-G [18], and UNICORE [12]. While these middleware provide convenient mechanisms for resource sharing and the needed security support, they demand the applications to be either developed with special library, or executed in application/service containers. Therefore, this approach does not seem to be suitable for most existing 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 construct grid platforms according to the needs of individual applications, so that they can harness remote computing resources without any modification.

Job schedulers (e.g., [3], [20], [21], etc.) have been a widely adopted approach to aggregate and share distributed resources. They support resource sharing by allowing remote clients to submit batch jobs to groups of machines that are managed by the schedulers. However, they generally fall short of being able to dynamically provide the suitable EE's to satisfy different application demands.

There have also been a number of application-specific solutions proposed by the industry such as the the Data Center Markup Language (DCML) [1]. While these solutions suit the needs of specific forms of resource sharing very well (e.g., access to databases, etc.), they do not provide the needed flexibility for supporting the existing applications. Run-time supports such as service provisioning [19][16], run-time adaption [15], and stub generation (e.g., [14]) fail to provide a transparent support; modification of existing applications is therefore needed.

To conclude, current research appear to be lacking in constructing platforms that an application *intends* to run on, which causes inconvenience and thus limits adoption. In this paper, we aim to provide a better solution to this problem.

3 Software Framework

In this section, we present the InstantGrid framework, which targets at efficient management and dissemination of the needed EE's to networked machines on-demand, according to the requirements of the target applications. The framework is designed to fulfill the following objectives.

Convenient system administration. Common management tasks associated with a particular EE include installation, configuration, software upgrade/update, and backup. If these have to be repeated for multiple EE's in a large number of machines (which might even be geographically distributed in grids), the induced cost and hassle could well offset the benefit of sharing the computing resources. Therefore, EE's should be centrally managed; the settings in the compute

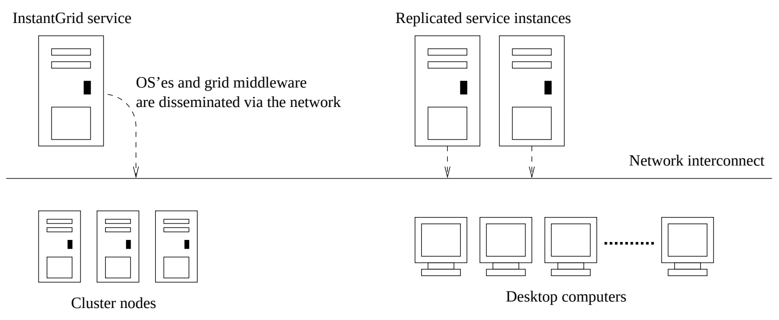
InstantGrid service
OS'es and grid middleware are disseminated via the network
Replicated service instances
Network interconnect
Cluster nodes
Desktop computers

Figure 1: InstantGrid Server and Compute Nodes as Clients

nodes being managed should be performed only once and kept simple.

Instant EE construction. InstantGrid aims to speed up the process in disseminating customized EE's from a central server to compute nodes for different grid applications. A running machine should only need to reboot in order to obtain a different EE from the InstantGrid service. Besides, InstantGrid should allow different implementation strategies for performance optimization in various network/system configuration.

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

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 for different types of platform ranging from dedicated cluster systems to home computers having idle cycles to share.

3.1 System Overview

The client/server paradigm is adopted in designing InstantGrid, which is depicted in Figure 1. As shown in the figure, all EE's are stored in and managed by an InstantGrid server, while the compute nodes obtain the EE's from the server on-demand to form the grid platform. It should be noted that the framework intends to allow replication of the InstantGrid service for better performance and reliability.

The framework consists of five system components (Figure 2): (1) application-centric software grouping, (2) proactive software configuration, (3) file sharing policy, (4) compute node storage management, and (5) EE dissemination service. The first four components collectively form the EE management model, which describes how EE's are managed in InstantGrid. The EE dissemination service is a low level system support for InstantGrid to handle the performance-critical dissemination process. We describe the design of each component in the following sub-sections. Due to the fact that the internal operations of the EE dissemination service are rather complicated in reality, we present the details of a reference implementation in the next section.

3.2 Application-Centric Software Grouping

In InstantGrid, an EE is defined as a collection of an UNIX-like OS, the supporting libraries and applications, grid middleware, cluster middleware, the user applications, and the user data. It essentially is a snapshot of all software components in a running system. In InstantGrid, these software components are grouped in an application-centric manner: there is an *EE specification* associated with each EE, which contains a list of software required by a specific application. EE's being grouped in this manner is advantageous since computing platforms are generally used by different users and

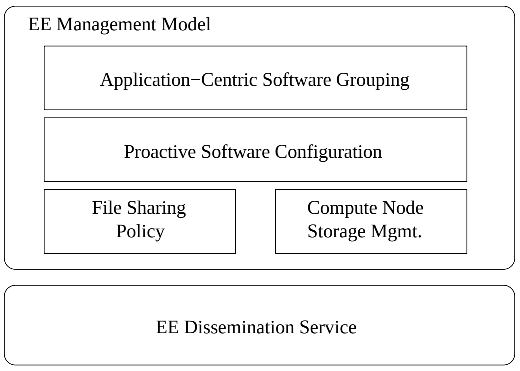
EE Management Model
Application−Centric Software Grouping
Proactive Software Configuration
File Sharing Policy
Compute Node Storage Mgmt.
EE Dissemination Service

Figure 2: System Architecture of InstantGrid

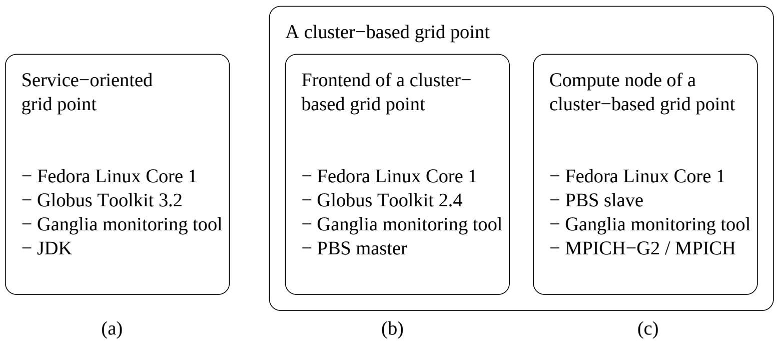
A cluster−based grid point
Service−oriented grid point
− Fedora Linux Core 1
− Globus Toolkit 3.2
− Ganglia monitoring tool
− JDK
Frontend of a cluster−based grid point
− Fedora Linux Core 1
− Globus Toolkit 2.4
− Ganglia monitoring tool
− PBS master
Compute node of a cluster−based grid point
− Fedora Linux Core 1
− PBS slave
− Ganglia monitoring tool
− MPICH−G2 / MPICH
(a)
(b)
(c)

Figure 3: Examples of EE's of Grid Points

hence their applications over time. Once administrators define the list of software in each EE, they can conveniently switch between EE's (i.e., to disseminate a different group of software according to the requirements of the next user), without dealing with the individual software components.

It should be noted that several EE's can be further grouped to reflect the requirement of some complex application scenario. For instance, a HPC cluster generally involves a frontend node acting as the scheduler, and the cluster nodes for computation; the frontend node has an EE which is different from that in the compute node. However, these two EE's are dependent on each other: the HPC cluster would not function if either is missing. It is therefore reasonable to manage them as a single unit. Figure 3 illustrates three possible groupings as examples: (a) is a service-oriented grid point, (b) a frontend node for HPC job submission, while (c) is a typical cluster node which processes jobs dispatched from the frontend node. (b) and (c) are grouped as a single *EE group* which defines the software requirement of a cluster-based grid point.

In order to avoid accidental removals of software or even an entire EE, there could be dependency checking routines incorporated in InstantGrid which check if a software/EE to be deleted is required by another EE.

3.3 Proactive Software Configuration

Traditionally, the OS and system software are installed and configured incrementally. In InstantGrid, by contrast, software belonging to the same EE have to be configured in the central server before disseminated to the compute nodes. Essentially, InstantGrid intends to maintain the (almost) ready-to-run version of an EE. This arrangement avoids the installation and configuration time during the grid point construction process.

Nevertheless, some software must perform *local* configuration. For instance, some grid middleware (e.g., Globus Toolkit) require host credentials such as certificates, which have to be handled locally at the respective nodes. In these cases, InstantGrid takes a “greedy” approach to configuration: it performs configuration tasks in the central server as much as possible, and leave minimum work to the compute nodes. This approach secures efficient composition of EE’s, which is especially useful to speed up the process to switch between two different EE’s.

3.4 Discriminative File Sharing Mechanisms

Disseminating the entire EE from the InstantGrid server to the compute nodes is challenging as the size of a typical EE could be in the order of gigabytes. Full replication is therefore impractical. However, to access all files through NFS is also inefficient since many files in an EE are frequently accessed; retrieving them through the NFS would result in poor runtime performance as well as heavy load at the file server. InstantGrid adopts a hybrid approach to the problem: it only replicates those that are frequently accessed (e.g., the files in the `/etc`), and leave the others in the file server for sharing via a network file system. The mechanisms adopted for individual directories are recorded in the EE specification. Detailed operation of this approach is presented in Section 4.4.

3.5 Compute Node Storage Management

InstantGrid allows the files being replicated to a compute node to be stored in the hard disk or entirely in the physical memory. If the files are stored in the hard disk, there is an extra option of I/O caching, which works as follows. Before a file is transferred from the InstantGrid server to a compute node, InstantGrid would first check if there is a local version of that file. If so, it would verify if it is up-to-date. File transfer would only be taken place if the file is found outdated.

One of the most useful features of InstantGrid is the support of in-memory execution. This allows any computer to participate in a grid (i.e., to contribute its computational resources) without affecting its local storage and hence its *default* purpose. A reference implementation of this feature relies on the Linux `ramdisk` support, which is described in details in Section 4.5.

To benefit from these support, administrators have to explicitly specify the storage policy for each compute node in the respective EE specifications.

3.6 EE Dissemination Service

The EE dissemination service comprises of several server (i.e., the InstantGrid server) and client (i.e., the compute nodes) components. The server components include (1) a network configuration service, (2) a request redirection module, and (3) a file server. The client components include (1) a network configuration module, (2) an EE service locater, (3) a network booting module, and (4) a file synchronization module.

Figure 4 illustrates the operations of these components, which work as follows. When a client machine boots up, the network configuration module sends or broadcasts a machine ID (e.g., the Ethernet Address, 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 of the client), 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 client.

Here starts the dissemination process: the network booting module at the client side obtains a boot loader from the EE dissemination 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

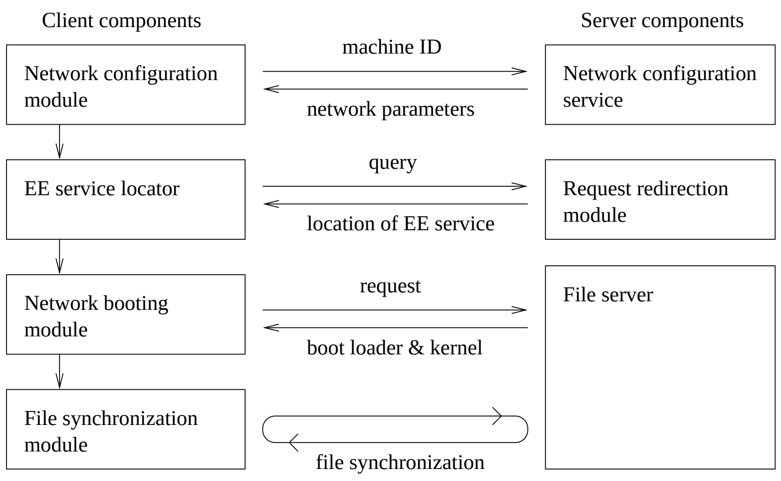

Figure 4: Client and Server Components

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 dissemination service and boot with it.

When the booting process finishes, the file synchronization module will obtain the EE specification, which is pre-defined by system administrators and specifies what the EE dissemination service will offer (i.e., which versions of boot loader/kernel, and which files are included in an OS image). The specification also dictates *how* the OS image (which includes shared libraries, binaries, and user data) could be retrieved from the central storage server(s). 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.

4 SLIM: Single Linux Image Management

We present a reference implementation of the EE dissemination service called SLIM, a network service for large-scale deployments of the Linux-based EE's in 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 does not affect or depend on any OS'es pre-installed in the hard disk
- Fast system recovery and backup. Client machines do not hold important user data; system administrators only need to perform backup at the central server
- 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.

We describe the work-flow of SLIM in LAN from Sections 4.1 to 4.4; and the options for local storage in Section 4.5.

4.1 System Overview

SLIM leverages legacy firmware/software supports for performing network configuration at and disseminating the EE's to remote clients. These supports include the pre-boot execution environment (PXE) [4], dynamic host configuration protocol (DHCP) [7], trivial file transfer protocol (TFTP) [5], `rsync` [11], and network file system (NFS) [2]. Apart

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

Table 1: Implementation Strategies of SLIM on LAN

from these, all software modules that we developed are merely a collection of *shell scripts*, which indicate the simplicity and portability of the EE dissemination service and the SLIM prototype. The implementation strategies of SLIM in LAN are summarized in Table 1.

There are three system processes running at the server side which collectively provide the EE dissemination service. These processes include the DHCP, TFTP, and NFS servers. The DHCP server sends the network configuration parameters (e.g., IP addresses, etc.) and the IP address of the TFTP server to the client machines. The TFTP server delivers the Linux kernel and a custom initial ramdisk (`initrd`) to the clients. The NFS server is the heart of the service, which hosts pre-installed Linux system images for different EE's. While these processes could execute in a single server machine in a small deployment, they are actually independent from each others and therefore could be distributed (and/or replicated) in multiple machines for higher performance and scalability.

We aim at minimizing the management effort at the client 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/user wishes to use another OS, he needs only to reboot the machine and repeat the booting process. Figure 5 gives an overview on the process of disseminating an EE to a client machine, which is discussed in details in the following sections.

4.2 EE Specification

SLIM relies on the EE specification managed by InstantGrid to identify the details of an EE to be disseminated. A specification includes (1) a list of software (hence files) in an EE; (2) a list of hardware profiles and drivers for different client machines; (3) a table which maps the client IP addresses to the corresponding hardware profiles; and (4) the file synchronization specification that determines which mechanisms are used to disseminate individual files. These lists and tables are stored as text files in the machine that hosts the SLIM service, which are retrieved when a client machine requests an EE during system boot-up.

4.3 Network Configuration and Booting

When a PC in a 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. The DHCP server of SLIM consults the EE specification to verify if the Ethernet address of the client is included in the list of hardware profiles. If that client is found eligible to obtain a specific EE, it would reply to the client with a `DHCP Offer` message, which includes the IP address being "leased" to that client, and the IP address of the SLIM TFTP server. Upon receipt of the IP addresses, the client 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 convenient transfer. `pxelinux`, after receiving the kernel and `initrd`, would uncompress `initrd` and load the image together with the

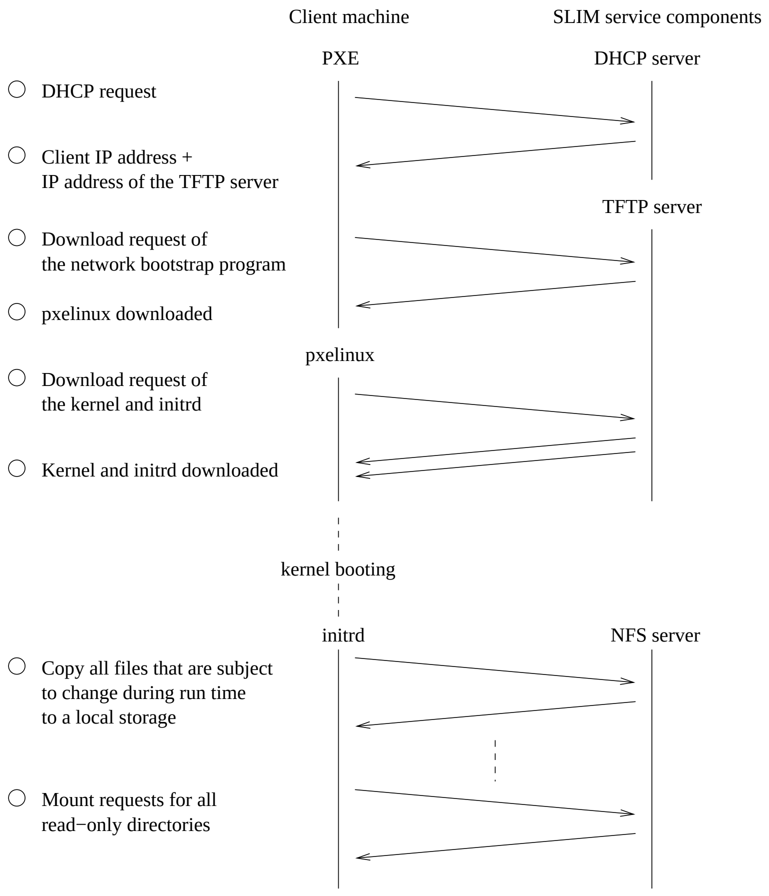

Figure 5: Process of Dissemination

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 that client machine
2. Mount the pre-installed Linux system image from the NFS server for file copy
3. Copy files to a local storage and mount the shared NFS volumes (more on this in the next section)
4. Switch to the file system on the local storage as the root (i.e., the “`/`”) file system
5. Start the `init` program to carry out normal system initialization process

4.4 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 NFS server. First is to copy the files from the server to local storage; second to access them through 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 retrievals 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.

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`	mainly read	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)

Table 2: Access Pattern of the Fedora Core 1 Image

NFS optimization. NFS brings convenience to both users and system administrators. However, it was found that the configuration options of the NFS affect performance considerably. This in fact stems from the unique *access pattern* of an OS image. Specifically, most files are read-only in an OS image. In a newly installed Fedora Core 1 image [8], for instance, 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 metadata consistency induces much unnecessary network traffic and thus decreases performance. In order to address this issue, we set the following NFS options: `ro`, which specifies a shared volume is 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 client. We found that these changes do improve performance considerably.

Cautious file transfers. 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. This, together with the success in optimizing the NFS performance, has encouraged us to use NFS quite aggressively: we choose to leave all read-only directories in the NFS server. Furthermore, all user data (i.e., their home directories) are also shared via NFS so that users can access their data from any machines, which is especially important in grid computing environments where the platforms might be expanded on demand. We believe these arrangements do achieve a good trade-off between the booting time and runtime performance. In fact, a later version of SLIM has been managing 250 desktop computers since 2002 and most users have not experienced significant performance degrade caused by the NFS activity.

4.5 Options for Local Storage

The files copied from the NFS server have to be stored locally. SLIM supports three options for local storage: (1) physical memory; (2) hard disk; and (3) hard disk with caching.

Physical memory. As shown in Table 2, the amount of files to be copied (i.e., the read-only ones) is relatively small (around tens of MB). Considering that modern PC systems are often equipped with 512 MB or even gigabytes of RAM, it is feasible to copy all 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 three advantages. First, it avoids the need to plan for the hard disk usage ahead, which is especially desirable in grid computing environments where a machine might execute different EE's over time. Second, it encourages adoption of grid technologies as the original OS/data stored in hard disks would not be altered. Third, idle computers could participate in grid computing easily.

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.

Hard disk with caching. 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. SLIM implements the caching feature by using the `rsync` protocol, which sends just the differences (if any) between the files across the network. Due to this reason, this option offers better performance than the others as network communication only involves version checking instead of the whole set of files.

5 Case Studies

In this section, we describe how InstantGrid and SLIM is deployed in our department. We also discuss the possibility of enabling grid computing at home via broadband networks.

5.1 InstantGrid/SLIM in HKU CS

InstantGrid has been managing 350+ cluster nodes in the Department of Computer Science, HKU. We have four PC clusters (350+ nodes) having different hardware configurations, which range from the Pentium II to Pentium IV, and a few AMD symmetric multiprocessors (SMPs). We found that managing the EE's for these computers for different applications induces much management hassles. Specifically, our research students, collaborators in the ApGrid [6] and HKGrid [10], and researchers in the engineering and science faculties constantly request to run experiments and demos in the cluster nodes; their requirements are usually very different from each other (e.g., different versions of Linux, JDK, C/Fortran compilers, proprietary libraries for scientific computation, etc). In the past, we divided the cluster nodes into multiple partitions for satisfying the different needs, but the main problem is poor utilization of some partitions. InstantGrid solved the problem very well as any EE can be disseminated according to the users' needs and the hardware requirements of target platform. Apart from the convenient system administration, a more important advantage is that we maintain a well-defined and consistent EE for our research collaborators, which is one critical factor to successful cross-domain collaborations.

Besides, SLIM itself has been managing the OS'es for another 250 machines which are mainly used for teaching or as students' desktops. The computers used for teaching encounter a problem similar to the HPC clusters. For instance, the requirement of the OS course differs substantially from that of the real-time system course. The students' desktops, by contrast, are generally installed with the Windows OS. Nevertheless, 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 Linux OS'es which execute entirely in the physical memory. Furthermore, since most of these computers are idle at nights, they could be restarted to form an ad-hoc HPC platform to process the jobs submitted by the users in our department and the other faculties. Although these cases do not relate directly to grid computing, they do demonstrate the flexibility and efficiency of SLIM (and hence the supported InstantGrid) in managing and disseminating multiple EE's.

5.2 InstantGrid Through Broadband Networks

InstantGrid over the broadband links could facilitate a wider adoption of grid computing since the general public could obtain the ready-to-run system software at home. Users could benefit from secure resource sharing such as file or cycle sharing. In addition, home computers have a very good potential in forming powerful platforms for massive computation as they generally have a lot of idle resources. While this type of computation has existed for years, grid technologies support standard mechanisms to perform remote resource access which encourage a much wider deployment. Advantages aside, however, disseminating EE's over slow links imposes new challenges as the performance of typical residential broadband networks is much lower than that of the LAN. To overcome the technical difficult is one of our future work.

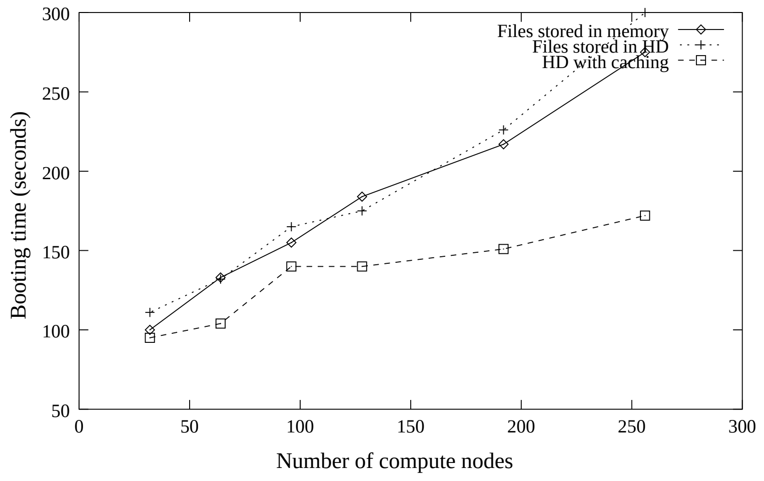

Figure 6: Booting Time of a Cluster-Based Grid Point

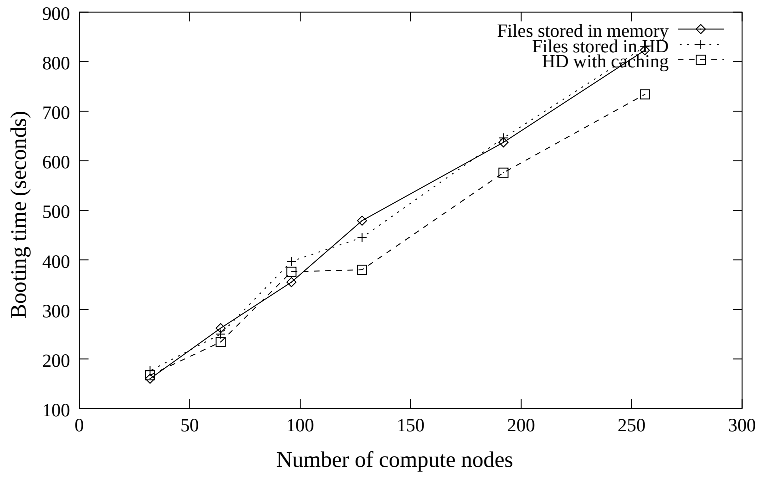

Figure 7: Booting Time of Standalone Service Grid Points

6 Experiments

We conducted two experiments with InstantGrid in the HKU CS Gideon cluster [9]. The cluster consists of 300 Pentium IV machines (we used 256 in the experiments) 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 inter-process communication. The management network is organized in a hierarchical manner: 13 24-port Fast Ethernet switches (each connects 22-24 nodes) are interconnected by a Gigabit Ethernet switch. All server processes of the InstantGrid service are hosted by a single Pentium IV machine with 512MB RAM and an IDE hard disk. We used InstantGrid to disseminate two different EE's to the compute nodes: one is for a cluster-based grid point (i.e. Figure 3(b) and (c)), the other is for standalone service grid points (Figure 3(a)). In both experiments, the Fedora OS is disseminated to the cluster nodes by copying the entire OS image except the following directories: `/bin`, `/lib`, `/sbin`, and `/usr`, which are mounted through the NFS. The Linux kernel and the `initrd` are transferred through TFTP, while the other files are copied by `rsync`. We tested the performance of InstantGrid under three storage options: files stored in physical memory (in a ramdisk of 50MB large); files stored in local hard disk; and local hard disk with caching. The last option enables the grid point to be constructed quickly if the files have been stored locally at the compute nodes before.

Figure 6 presents the booting time of a cluster-based grid point. The results are quite impressive since a 256-grid point can be constructed from scratch in five minutes (i.e., the case of ”Files stored in HD”). The good performance is mainly due to the proactive software configuration in InstantGrid, which shortens the dissemination time.

As shown in the figure, if the same EE has been stored in the hard disk before (i.e., ”HD with caching”), the construction can be as fast as about 3 minutes. Caching offers better results since the dissemination process only involves the transfer of the kernel and the queries for version checking. We were quite surprised to know that the performance of in-memory storage is similar to that of hard disk, which suggests the performance bottleneck lies more on the network than on the hard disk I/O. Nevertheless, the bottleneck can be relieved by replicating the InstantGrid service to multiple machines.

Figure 7 show the booting time of standalone service grid points. The time are considerably longer than that of the cluster-based grid point. The reason is that each compute node is treated as a standalone grid point, i.e., it requires a separate host certificate. The certificate request is generated during the dissemination process, and is sent to a dedicated certificate authority server for signature. Since this process has to be done sequentially (there is only one CA server for all certificate requests), the booting time became much longer. The results suggest that some software configuration processes are indeed time consuming, which have to be avoided or redesigned for more efficient deployments.

Overall, the experiments demonstrate that by using a single InstantGrid server, a 256-node cluster-based grid point can be constructed from 3 (with caching) to 5 (without caching) minutes, and that 256 standalone service grid points can be constructed in about 14 minutes. The results prove that InstantGrid is indeed efficient in constructing grid points.

7 Concluding Remarks

We have proposed the InstantGrid framework for on-demand construction of grid points. We have also presented a reference implementation and performance optimization strategies of the EE dissemination service. The experimental results show that InstantGrid is able to efficiently construct Linux-based grid points with commodity grid middleware.

Future work will focus on performance optimization of InstantGrid in WAN, which will be conducted in three dimensions. First, we will develop a custom file system which is able to mask the server/network failures and long network delay on WAN. In this file system, *partial file serving* will be implemented which allows the usage of a file (be it an executable or user data file) before it is completely fetched from the InstantGrid server, which improves the responsiveness of applications. Second, we aim to design negotiation protocols which can be used by a compute node to negotiate with the InstantGrid service for the optimal size and content of an EE based on the hardware capability of the compute node and the real-time network performance. Third, we plan to incorporate more “intelligence” in the discriminative file sharing policies, which takes the past usage history, users’ preferences, and the intended usage of a grid point to determine the optimal file sharing strategy.

8 Acknowledgment

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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*.

[17] 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/.*

[18] 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.

[19] 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.

[20] 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.

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