SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms

Xuxian Jiang, Dongyan Xu
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907
{jiangx, dxu}@cs.purdue.edu

Abstract

The Grid is realizing the vision of providing computation as utility: computation jobs can be scheduled on-demand in Grid hosts based on available computation capacity. In this paper, we study another emerging usage of Grid utility: the hosting of application services. *Different from a computation job, an application service such as e-Laboratory or on-line shopping has longer lifetime, and performs multiple jobs requested by its clients. A service Hosting Utility Platform (HUP) is formed by a set of servers in the Grid, and multiple application services will be hosted on the HUP.*

We present the design and implementation of SODA, a Service-On-Demand Architecture that enables on-demand *creation of application services on a HUP. With SODA, an application service will be created in the form of a set of* virtual service nodes*; each node is a virtual machine which is physically a 'slice' of a real host in the HUP. SODA involves both OS and middleware level techniques, and has the following salient capabilities: (1) on-demand service priming: the image of an application service as well as the OS on which it runs will be created on-demand and bootstrapped automatically; (2) better service isolation: services sharing the same HUP host are isolated with respect to administration, faults, attacks, and resources; (3) integrated service request management: for each service, a service switch will be created to direct client requests to appropriate virtual service nodes. Moreover, the application service provider can replace the default request switching policy with a service-specific policy.*

1 Introduction

The Grid is realizing the vision of providing computation as utility. Analogous to water and electricity, computation resources will be supplied on-demand to a computation job, and turned off when the job is completed. In this paper, we focus on another emerging usage of the Grid utility: the hosting of application services. Different from a computation job, an application service such as an e-Laboratory or an on-line business has longer lifetime, and performs multiple jobs for its clients. However, the application service provider may not wish to use its own resources to host the service. On the other hand, the Grid provides an excellent platform for this purpose: a service Hosting Utility Platform (HUP) can be formed by a set of servers in the Grid, and multiple services will be hosted on the HUP.

To reflect the utility vision, service hosting on a HUP should be *on-demand*. In other words, an application service should be dynamically created and automatically bootstrapped (and torn down), at the provider's request. For example, a bioinformatics institute wishes to provide a genome matching service to the research community, without using its limited IT resources. It can make a service creation call to a HUP, and the entire image of the genome matching service will be downloaded to and bootstrapped in the HUP. In addition, staff of the bioinformatics institute should be able to perform service monitoring and management, as if the service were hosted locally. To realize this picture, we need to address the following challenges.

The first challenge is the *virtualization* of services[1] and the *isolation* between them. Although sharing the same HUP, each service should appear to its provider as running in a dedicated environment, or more specifically, in a set of *virtual service nodes*: each node is a virtual machine which is physically a 'slice' of a real server in the HUP. Between services, isolation is desirable with respect to (1) administration - a service provider should have administrator privilege, but *only* within its own service; (2) fault and attack handling: a crash or security breach of one service should not affect other services running in the same HUP host; and (3) resources: resource allocation should be

[1]For the rest of the paper, a service means an application service if not otherwise specified.

guaranteed for each service, or more specifically, for the HUP host 'slices' that form the corresponding set of virtual service nodes.

The second challenge is the *on-demand* creation of services. Service providers should be able to make service creation requests, and images of application services will be transported to the HUP and get installed. Furthermore, to achieve service isolation described in the previous paragraph, each virtual service node will involve not only the service software, but also a virtual (guest) OS environment upon which the service software runs. It is challenging to enable automatic bootstrapping of both the service and the guest OS on top of the host OS in a HUP host. Unfortunately, current active service techniques [5, 34] are not adequate to handle such an 'active virtual machine' scenario.

In this paper, we present the design and implementation of SODA, a Service-On-Demand Architecture that enables the hosting of application services on HUPs. SODA integrates existing and novel techniques at both OS and middleware levels. The salient capabilities of SODA include: (1) on-demand service priming: the image of a service as well as its guest OS is created on-demand and bootstrapped automatically; (2) better service isolation: services sharing the same HUP host are isolated with respect to administration, fault/attack handling, and resources; (3) integrated service request management: for each service, a service switch will be created to direct client requests to appropriate virtual service nodes. Moreover, the service provider can replace the default request switching policy with a service-specific policy.

The rest of this paper is organized as follows: Section 2 gives an overview of SODA. Section 3 describes key entities of SODA. Section 4 presents SODA implementation. Section 5 presents the performance of application services in SODA. Section 6 compares SODA with related work. Section 7 concludes this paper.

2 Overview of SODA

SODA is illustrated in Figure 1. A HUP is formed by a collection of high-end hosts in the Grid. The HUP hosts may be connected by a local or wide area network. Currently, our testbed is in a departmental high-speed LAN. Customers of the HUP are *Application Service Providers* (ASPs). Each HUP host runs a *host OS* and a *SODA Daemon*. For the entire HUP, there is one *SODA Agent* and one *SODA Master*.

2.1 Service Virtualization and Isolation

Before describing the SODA entities, we first define and justify the *virtual service nodes* for service virtualization

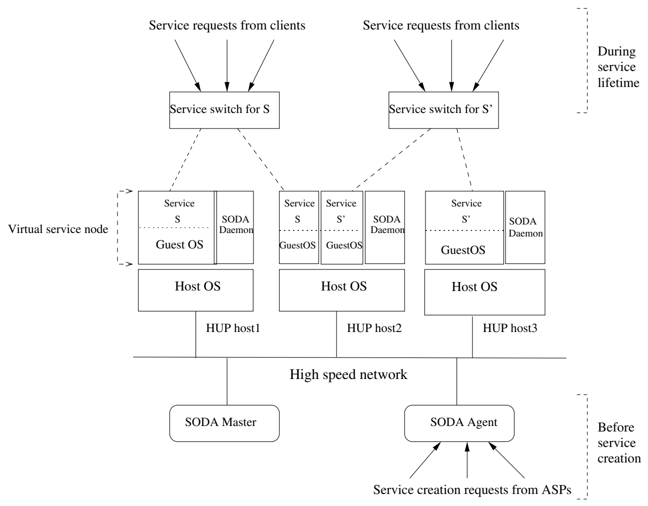


Figure 1. Overview of SODA

and isolation. Suppose an ASP requests the creation of service S. Based on resource requirement specified by the ASP, SODA will create a set of virtual machines for S. As shown in Figure 1, each virtual machine is called a virtual service node, which is physically a 'slice' of a HUP host. Each node runs a *guest OS* on top of the host OS; while service S runs on top of the guest OS. Moreover, an IP address is assigned to each virtual service node so that it can communicate like a physical server. A *service switch* for S is also created by SODA, directing each client request to one of the virtual service nodes for S.

To explain why there may be multiple virtual service nodes (each for a different service) on the same HUP host, we realize that the resource requirement of different services is highly diverse, while the resource availability of HUP hosts may not be uniform. Therefore, HUP host resources may not be fully utilized, if we choose one whole HUP host as the minimum unit of resource allocation.

For the 'guest OS/host OS' structure in each HUP host, we have the following justifications: (1) *Administration isolation*: It is desirable that each ASP has full administrator privilege within the corresponding virtual service nodes, so that the ASP can perform service-specific management tasks such as monitoring and software upgrade. However, if the administrator privileges of all ASPs are at the same (host OS) level, access control will become complicated and may lead to security holes. (2) *Fault/attack isolation*: following (1), if all services run at the same host OS level, any fault or security breach in one service will affect the host OS and therefore other services. For example, *ghttpd* [29] is a light-weight web server run by the root. However, one known attack to ghttpd is: a malicious packet is sent as an HTTP request, causing buffer overflow to bind a shell on a certain port. Then the attacker can remotely log in using the port, and run a remote shell! With SODA, since the root that runs ghttpd is the root of the *guest OS*, not the host OS, the attack will *not* affect the host OS as well as other services. (3) Existing OS techniques of resource reservation and scheduling can only achieve resource isolation. They are not adequate to achieve service isolation.

2.2 Key Entities in SODA

Functions of key SODA entities are summarized as follows.

- **SODA Agent** is a middleware-level entity serving as the interface between the ASPs and the HUP. It accepts service creation requests and performs other administrative tasks such as billing.
- **SODA Master** is a middleware-level entity coordinating the service creation activities across the HUP. More specifically, SODA Master determines the set of virtual service nodes for each service creation request and coordinates the service priming process.
- **SODA Daemon** is a system-level entity running in each HUP host. It performs the downloading of application service images and the bootstrapping of virtual service nodes.
- **Service switch** is an application-level entity created by SODA for each service. After the service has been bootstrapped, the service switch will accept client requests and directs them to appropriate virtual service nodes.

3 SODA Entities and Operations

In this section, we describe in detail the key SODA entities and their interactions, by following the creation of a service. Current limitations of SODA will also be discussed. To request the hosting of a service S, the ASP needs to prepare: (1) the image of service S, including the executables and data files, properly organized in a file system. The image should be stored in a machine owned by the ASP; (2) the resource requirement of S. Currently, the requirement is specified as a tuple $< n, M >$, meaning that the hosting of service S requires n machines of configuration M - M is a tuple indicating the types and amounts of resources. An example of M is shown in Table 1. The resource requirement specification is the result of off-line QoS/resource profiling [13], which is out of the scope of this paper.

Type of resource	*Amount of resource*
CPU	512MHz
Memory	256MB
Disk	1GB
Bandwidth	10Mbps

Table 1. Example of machine configuration M in resource requirement $< n, M >$

3.1 SODA Agent

The ASP then makes a service creation request to the SODA Agent. The request contains the service image location as well as the resource requirement. As the interface between ASPs and the HUP, the SODA Agent authenticates the ASP and passes the request to the SODA Master. After the service creation (to be described) is completed, the SODA Agent will reply to the ASP with information about the virtual service nodes created for S.

3.2 SODA Master

Upon receiving the service creation request, the SODA Master checks if the resource requirement of S can be satisfied by current HUP resource availability. The SODA Master collects resource information from SODA Daemons running in each HUP host. If the resource requirement cannot be satisfied, a request failure will be reported. Otherwise, service S will be admitted; and the SODA Master will identify a number of HUP host 'slices' to form the set of virtual service nodes for S. The SODA Master will then contact the SODA Daemons running in the selected HUP hosts to initiate the service priming process. After service priming, the SODA Master will create a service switch for S.

The SODA Master maps the service resource requirement $< n, M >$ to n' ($n' \leq n$) virtual service nodes. Our current implementation assumes that (1) service S is *fully* replicated in each virtual service node and (2) the minimum granularity of each virtual service node is one machine instance M - in other words, the capacity of one virtual service node is either one M or a multiple of M. Our assumptions simplify the resource allocation algorithm of the SODA Master. However, two problems exist: (1) resource aggregation: if more than one Ms are mapped to one virtual service node, this node may require *less* amount of a certain resource (such as disk space) than in the multiple Ms. (2) the slow-down effect: since each virtual service node is a virtual machine running on the host OS, there will be a *slow-down* in both processing and network transmission. It is a challenging problem (and our on-going work) to determine the factors of resource aggregation and slow-down. Before these problems are solved, we use a conservative estimation of these factors in the SODA implementation[2].

3.3 SODA Daemon

A SODA Daemon is running in each HUP host as a host OS process. It reports resource availability to the SODA Master. And it performs *service priming*, i.e. the creation of a virtual service node, at the command of the SODA Master. Upon receiving the command to create a virtual service node, the SODA Daemon will contact the underlying host OS and make resource reservations for the virtual service node. After reserving a 'slice' of the HUP host, the SODA Daemon will download the service image from the location specified by the ASP (Section 3.1), and bootstrap the virtual service node (first the guest OS, then the service). Note that once the service is started, the SODA daemon will *not* interfere with the interactions between the virtual service node and the host OS.

During the bootstrapping, the SODA Daemon will also assign an IP address to the virtual service node, so that it can communicate like a real server. In the SODA implementation, this is enabled by a *bridging module* running in the host OS, which acts as a transparent bridge connecting all virtual service nodes in the HUP host[3]. The SODA Daemon will return this IP address to the SODA Master, which will collect the IP addresses of all virtual service nodes created for S.

3.4 Service Switch

After the SODA Daemons have finished service priming, the SODA Master will create a service switch for S, based on the information returned by the SODA Daemons. Co-located in one of the virtual service nodes of S, the service switch will accept and direct each client request to one of the virtual service nodes. The service switch enforces a default request switching policy, which can be *replaced* with a service-specific policy by the ASP.

Inside the service switch, a *service configuration file* is created and maintained by the SODA Master. The file records (1) the IP address and (2) the relative capacity of each virtual service node of S. If the ASP decides to *resize* the service capacity, the SODA Master will either adjust the resources in the current virtual service nodes, or add/remove virtual service node(s). In either case, the service configuration file will be updated by the SODA Master to reflect the changes.

[2]More specifically, we set the slow-down factor to be 1.5 and we assume no resource aggregation.

[3]However, if the scarcity of IP addresses becomes a problem, we will adopt the technique of *proxying* instead of bridging, so that a virtual service node can still communicate with a reserved IP address.

Via SODA, service S is now created as the set of virtual service nodes and the service switch.

3.5 Discussion

The current design of SODA has the following limitations:

- A slow-down is inevitably associated with the guest OS/host OS structure. As a result, the CPU and network bandwidth requirement has to be 'inflated' during resource allocation. This is the price for service isolation, especially with respect to service administration and fault recovery. With the rapid advances in high performance processors, such a price is expected to be more affordable in the near future.

- SODA is *not* proposed as a solution to service security or fault tolerance. It only helps to 'jail' the impact of fault or attack within one service instead of 'saving' the service. We further note that the service isolation achieved by SODA is *not* absolute. For example, if a service is DDoS-attacked, its service switch will be inundated with requests, affecting other virtual service nodes in the same HUP host and therefore violating the service isolation.

- Currently, SODA only supports fully replicated services, i.e. the same service image is mapped to every virtual service node. However, a more flexible service image mapping is desirable, in order to accommodate a wider spectrum of services - for example, a partitionable service [25] where *different* service components are mapped to different virtual service nodes.

- SODA currently focuses on a local HUP rather than a wide-area HUP. One way to construct a wide-area HUP is to *federate* multiple local HUPs, each having its own SODA Agent and Master. However, we will need to address challenges including autonomous management (under different ownership), distributed monitoring [33], and platform heterogeneity of multiple HUPs.

The purpose of this paper is to demonstrate the architecture of SODA and its prototype implementation, rather than the final design and implementation. Therefore, we do not present solutions to the above limitations which are part of our on-going work.

4 SODA Implementation

In this Section, we present the implementation of a SODA prototype. Some initial performance of SODA will also be presented. There are currently two HUP hosts in our testbed, coded *seattle* and *tacoma*, respectively. *seattle* is a Dell PowerEdge server with a 2.6GHz Intel Xeon processor and 2GB RAM, while *tacoma* is a Dell desktop PC with a 1.8GHz Intel Pentium 4 processor and 768MB RAM. In addition, there are a number of laptop and desktop PCs running as the SODA Agent, SODA Master, and service clients. All machines are connected by a 100Mbps LAN.

4.1 SODA API

SODA provides APIs for service creation, tear-down, and resizing. The SODA Agent accepts these calls and passes them to the SODA Master after proper authentication. *SODA_service_creation* allows the ASP to specify service name, location of service image, and resource requirement $< n, M >$ (Section 3). *SODA_service_teardown* allows the ASP to request the tear-down of a service. *SODA_service_resizing* may be called by the ASP to resize the service capacity based on a new resource requirement $< n_{new}, M >$.

4.2 Host OS and Guest OS

Currently, SODA supports Linux as the host OS of each HUP host. For the guest OS, we leverage and extend an open source project called UML [18], or User-Mode Linux. Unlike other virtual machine techniques such as VMWare [3, 35], a UML runs directly in the unmodified *user space* of the host OS; and processes within a UML (the guest OS) will be executed in the virtual service node exactly the same way as they would be executed in a native Linux machine. A special thread is created to intercept the system calls made by all process threads of the UML, and redirect them into the host OS kernel. Meanwhile, the host OS has a separate *kernel space*, eliminating any security impact caused by the individual UMLs.

The original UML only provides limited support for resource isolation: For *memory*, a memory usage limit can be specified as a parameter when a UML is started. Then the memory consumption of the corresponding virtual service node (both the UML and the service) will not exceed this limit. However, the original UML does *not* support the isolation of other resources such as *CPU* and *network bandwidth*. To solve this problem, we enhance the *Linux host OS* with the following:

- *CPU isolation* We have implemented a coarse-grain CPU proportional sharing scheduler, which enforces the CPU share allocated to each virtual service node. The CPU share is determined by the SODA Master when the corresponding service is admitted. Within one virtual service node, all processes bear the same user (service) id. The CPU scheduler in the host OS then enforces proportional CPU sharing among all processes, based on their userids. Performance of CPU isolation will be presented in Section 5.

- *Network bandwidth isolation* We are implementing a traffic shaper inside the Linux host OS, which enforces the outbound bandwidth share allocated to each virtual service node. Recall that each virtual service node has its own IP address. Therefore, the traffic shaper achieves bandwidth isolation based on the IP addresses of outgoing packets generated by different virtual service nodes.

4.3 On-Demand Service Priming

The original UML system does *not* support on-demand and automatic service priming. Therefore, we implement the SODA Master and SODA Daemon to perform this task.

Active service image downloading The first step of service priming is to download the service image from the specified location. We assume that the ASP has properly packaged the service image (including the executable and the data files) using RPM, so that it is organized into a file system with one root. After receiving the service priming command from the SODA Master, the SODA Daemon on each selected HUP host will download the service image using HTTP/1.1. The downloading time depends on the network connection between the service image repository and the HUP host. We have measured the downloading time for service images of different sizes within the 100Mbps LAN. As expected, the downloading time grows linearly with the size of the service image.

Automatic bootstrapping of virtual service node The second step of service priming is the bootstrapping of UML (guest OS) and the application service. To achieve fast bootstrapping, the SODA Daemon first performs a *customization* of the Linux system services to be started in the UML. SODA Daemon tailors the root file system of the UML[4] by retaining only the Linux system services (in the /etc/ directory) required by the application service; it also checks their dependencies to ensure that only the necessary libraries are included. The customized root file system is light-weight and reconfigurable - in many cases it can be mounted in RAM disk for fast bootstrapping. Finally, the SODA Daemon executes a command (provided by the UML system) to start the UML; the UML then starts the application service.

To demonstrate the highly efficient bootstrapping of virtual service nodes in SODA, we measure the bootstrapping time using four different application services as

[4]Note that the application service image is also part of the root file system.

App. service	*Linux configuration*	*Image size*	*Time (seattle)*	*Time (tacoma)*
S_I	rootfs_base_1.0	29.3MB	3.0 sec.	4.0 sec.
S_{II}	root_fs_tomrtbt_1.7.205	15MB	2.0 sec.	3.0 sec.
S_{III}	root_fs_lfs_4.0	400MB	4.0 sec.	16.0 sec.
S_{IV}	root_fs.rh-7.2-server.pristine.20021012	253MB	22.0 sec.	42.0 sec.

Table 2. Service bootstrapping time for four different application services

Directive	*IP address*	*Port number*	*Capacity*
BackEnd	128.10.9.125	8080	2
BackEnd	128.10.9.126	8080	1

Table 3. A sample service configuration file created by the SODA Master after service priming

shown in Table 2: they differ in the number and type of Linux system services needed. Each of S_I, S_{II} and S_{III} requires a tailored (and different) subset of Linux system services, while S_{IV} requires a full-blown Linux server (rh-7.2-server-pristine). The bootstrapping time on both HUP hosts is very short, compared with the worst-case of S_{IV}. Note that the bootstrapping time is not solely dependent on the service image size, it is more dependent on the number and type of Linux services needed.

Dynamic configuration for internetworking To enable internetworking for virtual service nodes, each SODA Daemon maintains a pool of IP addresses to be assigned to the virtual service nodes running in this HUP host. For different HUP hosts, their pools of IP addresses must be disjoint. After assigning an IP address to a UML (i.e. virtual service node), the SODA Daemon will notify the bridging module inside the host OS (Section 3.3) of the new 'UML-IP' mapping, so that the bridging module will correctly forward packets from/to the new virtual service node.

Service request switching and service resizing After the virtual service nodes are created, the SODA Master will create a service switch which contains a *service configuration file*. A sample service configuration file is shown in Table 3. It records the IP address, port number, and capacity of each virtual service node. The capacity is relative to the number of machine instances M (in $< n, M >$) mapped to this virtual service node. In Table 3, the resource requirement of the service is $< 3, M >$, and is provided by two virtual service nodes with capacity of $2M$ and M, respectively.

5 Application Service Performance in SODA

In this section, we present the performance of application services created in SODA. The first service (S_I in Table 2) is a simple web content service which provides a static dataset to clients. To illustrate service isolation, the second service (S_{II} in Table 2) is deliberately a 'dangerous' service called *honeypot* [1]: it provides a vulnerable 'victim' server to be attacked by malicious clients. Such an *attack emulation* service is useful to the understanding and prevention of attacks in the real world. The two services are created in the two HUP hosts (Figure 2): the web content service has two virtual service nodes: the one in *seattle* has twice as much capacity as the one in *tacoma*. The honeypot service has one virtual service node in *seattle*.

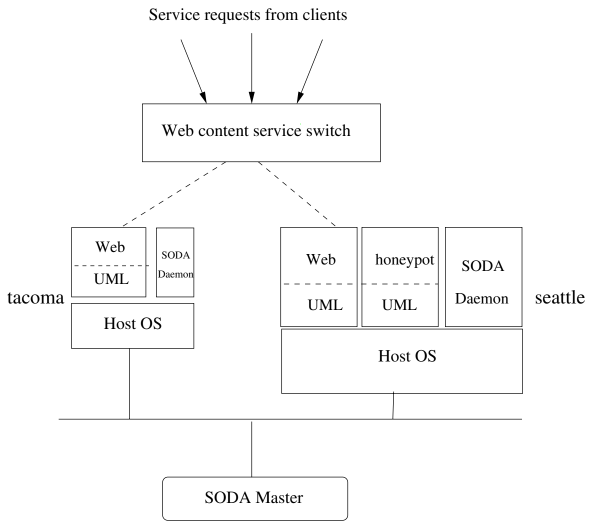
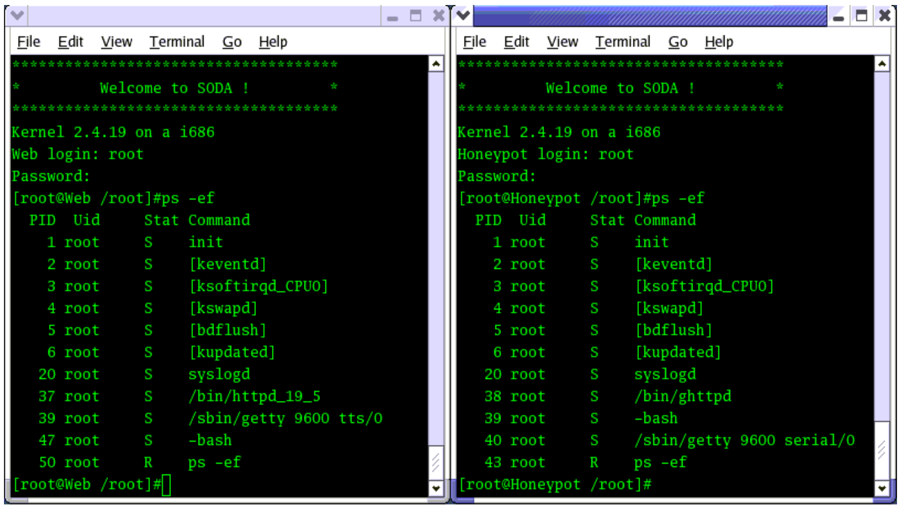


Figure 2. Creation of web content service and honeypot service in the SODA testbed

Attack isolation In this experiment, the honeypot service is constantly attacked and crashed. However, the web content service is *not* affected. Figure 3 is a screenshot of HUP host *seattle* showing their co-existence: The two windows correspond to the two virtual service nodes (left: web content service, right: honeypot service). The 'ps -ef' commands are executed under their own guest OSes



```
*************************************
*        Welcome to SODA !          *
*************************************
Kernel 2.4.19 on a i686
Web login: root
Password:
[root@Web /root]#ps -ef
  PID  Uid     Stat Command
    1 root     S    init
    2 root     S    [keventd]
    3 root     S    [ksoftirqd_CPU0]
    4 root     S    [kswapd]
    5 root     S    [bdflush]
    6 root     S    [kupdated]
   20 root     S    syslogd
   37 root     S    /bin/httpd_19_5
   39 root     S    /sbin/getty 9600 tts/0
   47 root     S    -bash
   50 root     R    ps -ef
[root@Web /root]#
```

```
*************************************
*        Welcome to SODA !          *
*************************************
Kernel 2.4.19 on a i686
Honeypot login: root
Password:
[root@Honeypot /root]#ps -ef
  PID  Uid     Stat Command
    1 root     S    init
    2 root     S    [keventd]
    3 root     S    [ksoftirqd_CPU0]
    4 root     S    [kswapd]
    5 root     S    [bdflush]
    6 root     S    [kupdated]
   20 root     S    syslogd
   38 root     S    /bin/ghttpd
   39 root     S    -bash
   40 root     S    /sbin/getty 9600 serial/0
   43 root     R    ps -ef
[root@Honeypot /root]#
```

Figure 3. Screenshot showing the co-existence of virtual service nodes for the web content service (left) and for the honeypot service (right)

(UMLs). More specifically, an *httpd* (httpd_19_5) daemon is running for the web content service; while a *ghttpd* (ghttpd-1.4) daemon is running as the current 'victim' server in honeypot[5].

Service request switching and load balancing We use the web content service to demonstrate service load balancing achieved by the service switch. The request switching policy is weighted round-robin, with the weights reflecting the capacity of the two virtual service nodes in *seattle* and *tacoma*, respectively. We measure the average request response time achieved by each virtual service node; and the measurement is repeated under six different dataset sizes. The requests are generated by machines in the same LAN using *siege*, an HTTP request generation program; and we reduce the request arrival rate with the increase in dataset size. We observe that the requests served by the node in *seattle* is approximately twice as many as those served by the node in *tacoma*. More importantly, the request response time achieved by the two nodes are approximately the same, as shown in Figure 4. The results show that by enforcing an appropriate request switching policy, the service switch will achieve load balancing among virtual service nodes for a service.

Note that the web content service is only a simple example. For a more complex service, the ASP may need to replace the default request switching policy with a service-specific policy. Thanks to the service isolation in SODA, even if the service-specific policy is ill-behaving, it will not affect other services hosted in the HUP.

[5]As described in Section 2.1, ghttpd server contains a remotely exploitable buffer overflow which allows an attacker to gain ghttpd's privilege.

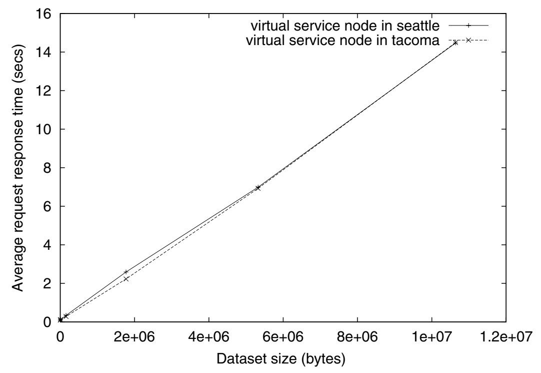
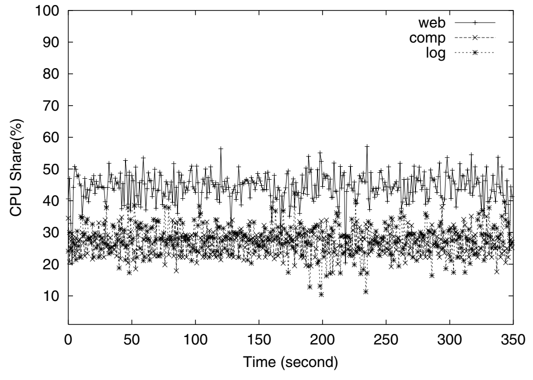
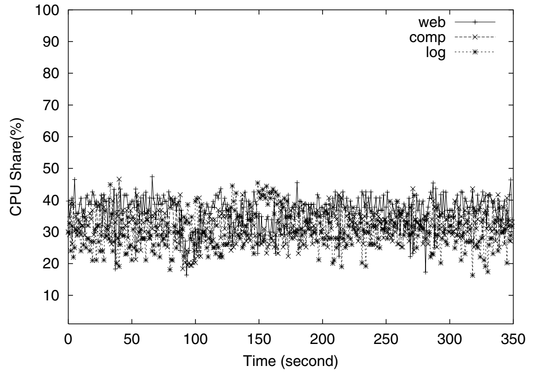


Figure 4. Average request response time of the web content service achieved by the two virtual service nodes in *seattle* and *tacoma*, respectively - the former serves approximately twice as many requests as the latter, under each dataset size

Resource isolation To demonstrate the resource (CPU) isolation achieved by SODA, another experiment is performed: we create two additional virtual service nodes *comp* and *log* in *tacoma*, besides the one for web content service (*web*). *comp* performs computation-intensive jobs with infinite loop of dummy arithmetic operations. *log* performs logging via continuous disk writes. Each of the three virtual service nodes (*web, comp* and *log*) is allocated an *equal* share of the CPU. However, their loads are *higher* than their respective shares. Under this loaded condition, we measure the actual CPU shares of the three virtual service nodes, as shown in Figure 5. Figure 5(a) shows the CPU shares (versus time) when the host OS is the *unmodified* Linux; while Figure 5(b) shows the CPU shares when the host OS is our enhanced Linux with the CPU proportional sharing scheduler. We observe that the 'equal-share' isolation between the virtual service nodes is better enforced by our enhanced host OS.



(a) Host OS: unmodified Linux



(b) Host OS: Linux with our CPU sharing scheduler

Figure 5. CPU shares (versus time) of the three virtual service nodes *web*, *comp* and *log*

Slow-down effect Finally, we measure the processing and transmission slow-down due to the guest OS/host OS structure. We first go to the 'source' of slow-down: the system calls. Table 4 compares the time (in clock cycles) to complete a system call in a UML and in the host OS. The results indicate that the guest OS/host OS structure does incur significant overhead in system call handling. We then measure the *application-level* slow-down using the web content service: Under the same service load[6], we run

System call	*in UML*	*in host OS*
dup2	27276	1208
getpid	26648	1064
geteuid	26904	1084
mmap	27864	1208
mmap_munmap	27044	1200
gettimeofday	37004	1368

Table 4. Measuring slow-down at system call level (clock cycles)

the web content service in three different scenarios: (1) in one virtual service node with service switch; (2) *directly* on the host OS with service switch; and (3) *directly* on the host OS without service switch. In all three scenarios, there is *no* other service load in the system. The request response time is compared in Figure 6. We again observe a slow-down incurred by the virtual service node. However, the slow-down factor is much lower than the one indicated in Table 4; and it remains approximately the same under different dataset sizes. Although this experiment suggests that the slow-down factor is quite acceptable at the application level, more extensive experiments are needed before a general conclusion can be drawn.

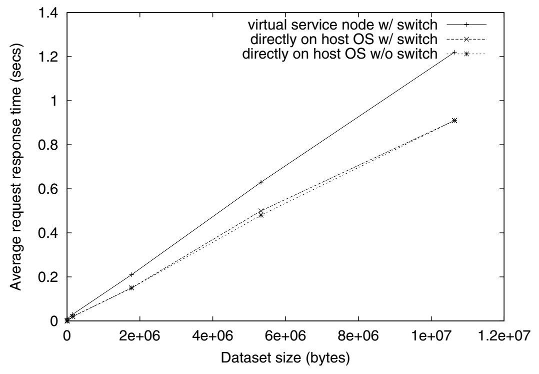


Figure 6. Measuring slow-down at application level (request response time)

6 Related Work

The Grid reflects the philosophy of utility computing. A number of projects have proposed the usage of Grid resources as *computation* utility, such as Globus [22], Condor [19], Legion [24], NetSolve [7], Harness [27] and Cactus [4]. Meanwhile, the Grid resources can also be used as a data *storage and management* utility, such as in the

[6]The service load in this experiment is lighter than in the previous experiments.

Storage Resource Broker [9], NeST [10], the Data Grid [14], and OceanStore [26]. Industry efforts parallel those of the academia, such as Oceano [6] at IBM and Planetary Computing [2] at HP. In this paper, we focus on *application service hosting* as another emerging usage of Grid utility.

Recently, the vision of integrating Grid and Web service concepts and technologies has been proposed as the Open Grid Services Architecture (OGSA) [22, 23]. With efforts in OGSA implementation and standardization, application services are expected to be widely deployed and achieve higher interoperability. While OGSA addresses higher level issues such as service description, discovery and composition; our work focuses on underlying issues such as service hosting and isolation, complementing the OGSA efforts.

SODA is related to active or on-demand services. In active services, the executable of a service can be dynamically downloaded and started where it is needed (for example, to perform QoS adaptation or to handle increasing load). Examples include the Berkeley Active Service Framework [5], WebOS [34], Darwin [12] and the Adaptive Service Grid [36]. Different from active services, SODA supports on-demand creation of both services and guest OSes on top of which the services will run. As a result, services created by SODA enjoy better isolation.

Resource isolation has been realized by existing techniques of resource reservation, allocation, and scheduling, such as those in QLinux [32], Resource Kernel [30], GARA [21], and Virtual Services [31]. Resources such as CPU, bandwidth, and memory can be reserved and allocated to different processes, users, or service classes; and the allocation is enforced by various resource scheduling algorithms. Resource partitioning is also proposed for server clusters [8] and overlay networks [11], where slices of resources in multiple hosts are reserved for different applications or services. However, these works do not address other aspects of isolation such as administration and fault handling; nor do they support on-demand service creation.

The concept of virtual machines has been proposed and realized. Examples include Berkeley NOW [17] and High Performance Virtual Machines (HPVM) [15, 16]. Their goal is to *aggregate* the computing power of commodity machines (interconnected using high-performance communication techniques) and create a virtual high-performance supercomputer. On the other hand, our work involves the *partition* of physical servers into multiple virtual service nodes to host different services, each running on a guest OS. Recently, host and network virtualization has received tremendous attention: Denali [37] is a isolation kernel providing isolation between Internet services on shared hardware. SODA shares the same goal. In addition, SODA enables on-demand service priming and integrated service request switching among *multiple* virtual service nodes. The Cluster-On-Demand system [28] enables on-demand creation of virtual clusters. However, each cluster node is a real machine primed with a *host* OS based on bare hardware; while each virtual service node created by SODA is a virtual machine primed with *guest* OS and application service, based on the host OS. Finally, Virtuoso [20] realizes Grid computing in virtual machines, paralleling our efforts in service hosting in virtual service nodes. [20] also provides a view on resource management and virtual networking under virtual machines.

7 Conclusion

We have presented the design and implementation of SODA, a Service-On-Demand Architecture for hosting utility platforms. With SODA, ASPs can outsource application service hosting by calling the SODA APIs, yet they are able to perform service monitoring and administration as if the services were hosted in-house. SODA involves both OS and middleware level techniques. Our initial experiments show that SODA achieves highly efficient on-demand service priming and satisfactory service isolation. We plan to expand our testbed and perform more extensive performance evaluation of SODA.

8 Acknowledgments

We would like to thank the anonymous referees for their constructive comments and suggestions. This work was supported in part by a grant from the e-Enterprise Center at Discovery Park, Purdue University.

References

[1] honeynet. *http://www.honeynet.org*.

[2] Planetary Computing, HP Labs. *http://www.hpl.hp.com/news/2001/oct-dec/planetary.html*.

[3] VMWare. *http://www.vmware.com*.

[4] G. Allen, T. Dramlitsch, I. Foster, N. Karonis, M. Ripeanu, E. Seidel, and B. Toonen. Supporting Efficient Execution in Heterogeneous Distributed Computing Environments with Cactus and Globus. *Proceedings of Supercomputing 2001*, Nov. 2001.

[5] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. *Proceedings of ACM SIGCOMM '98*, Sept. 1998.

[6] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, and M. Kalantar. Oceano: SLA based Management of a Computing Utility. *Proceedings of IFIP/IEEE Intl. Symp. on Integrated Network Management*, May 2001.

[7] D. Arnold, H. Casanova, and J. Dongarra. Innovation of the NetSolve Grid Computing System. *Concurrency and Computation: Practice and Experience*, 2003.

[8] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. *Proceedings of the ACM SIGMETRICS 2000*, June 2000.

[9] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. *Proceedings of IBM CASCON'98*, Nov. 1998.

[10] J. Bent, V. Venkataramani, N. Leroy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. *Proceedings of IEEE HPDC-11*, July 2002.

[11] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: an Overlay Peer Utility Service. *Proceedings of IEEE OPENARCH 2002*, July 2002.

[12] P. Chandra, Y. Chu, A. Fisher, J. Gao, C. Kosak, E. Ng, P. Steenkiste, E. Takahashi, and H. Zhang. Darwin: Customizable Resource Management for Value-Added Network Services. *IEEE Network*, 15(1), 2001.

[13] F. Chang, A. Itzkovitz, and V. Karamcheti. User-level Resource-Constrained Sandboxing. *Proceedings of the 4th USENIX Windows Systems Symposium*, Aug. 2000.

[14] A. Chervenak, I. Foster, C. S. C. Kesselman, and S. Tuecke. The Data Grid: Towards an Architecture for the Distributed Management and Analysis of Large Scientific Data Sets. *Proceedings NetStore'99*, Oct. 1999.

[15] A. Chien, M. Lauria, R. Pennington, M. Showerman, G. Iannello, M. Buchanan, K. Connelly, L. Giannini, G. Koenig, S. Krishnamurthy, Q. Liu, S. Pakin, and G. Sampemane. Design and Evaluation of an HPVM-based Windows NT Supercomputer. *Journal of High-Performance Computing Applications*, 13(3), 1999.

[16] A. Chien, S. Pakin, M. Lauria, M. Buchanan, K. Hane, L. Giannini, and J. Prusakova. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Mar. 1997.

[17] D. Culler, A. Arpaci-Dusseau, R. Arpaci-Dusseau, B. Chun, S. Lumetta, A. Mainwaring, R. Martin, C. Yoshikawa, and F. Wong. Parallel Computing on the Berkeley NOW. *Proceedings of the 9th Joint Symposium on Parallel Processing*, June 1997.

[18] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[19] D. Epema, M. Livny, R. van Dantzig, X. Evers, and J. Pruyne. A Worldwide Flock of Condors: Load Sharing among Workstation Clusters. *Future Generation Computer Systems*, 12, 1996.

[20] R. Figueiredo, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. *Proceedings of IEEE ICDCS 2003, to appear*, May 2003.

[21] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. *Proceedings of IEEE IWQoS'99*, June 1999.

[22] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. *Open Grid Service Infrastructure WG, Global Grid Forum*, June 2002.

[23] D. Gannon, R. Bramley, G. Fox, S. Smallen, A. Rossi, R. Ananthakrishnan, F. Bertrand, K. Chiu, M. Farrellee, M. Govindaraju, S. Krishnan, L. Ramakrishnan, Y. Simmhan, A. Slominski, Y. Ma, C. Olariu, and N. Rey-Cenvaz. Programming the Grid: Distributed Software Components, P2P and Grid Web Services for Scientific Applications. *Cluster Computing*, 5(3), 2002.

[24] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An Operating System for Wide-Area Computing. *IEEE Computer*, 32(5), 1999.

[25] A. Ivan, J. Harman, M. Allen, and V. Karamcheti. Partitionable Services: A Framework for Seamlessly Adapting Distributed Applications to Heterogeneous Environments. *Proceedings of the IEEE HPDC-11*, July 2002.

[26] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. *Proceedings of ASPLOS 2000*, Nov. 2000.

[27] M. Migliardi and V. Sunderam. Heterogeneous Distributed Virtual Machines in the Harness Metacomputing Framework. *Proceedings of IEEE HCW'99*, Apr. 1999.

[28] J. Moore and J. Chase. Cluster On Demand. *Duke University Technical Report CS-2002-07*, May 2002.

[29] G. Owen. ghttpd. *http://gaztek.sourceforge.net/ghttpd*.

[30] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource Kernels: A Resource-Centric Approach to Real-Time Systems. *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, Jan. 1998.

[31] J. Reumann, A. Mehra, K. Shin, and D. Kandlur. Virtual Services: A New Abstraction for Server Consolidation. *Proceedings of USENIX 2000 Annual Technical Conference*, June 2000.

[32] V. Sundaram, A. Chandra, P. Goyal, and P. Shenoy. Application Performance in the QLinux Multimedia Operating System. *Proceedings of the Eighth ACM Conference on Multimedia*, Nov. 2000.

[33] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, and M. Thompson. A Monitoring Sensor Management System for Grid Environments. *Proceedings of IEEE HPDC-9*, Aug. 2000.

[34] A. Vahdat, T. Anderson, M. Dahlin, D. Culler, E. Belani, P. Eastham, and C. Yoshikawa. WebOS: Operating System Services For Wide Area Applications. *Proceedings of IEEE HPDC-7*, July 1998.

[35] C. Waldspurger. Memory Resource Management in VMware ESX Server. *Proceedings of USENIX OSDI 2002*, Dec. 2002.

[36] J. Weissman and B. Lee. The Service Grid: Supporting Scalable Heterogeneous Services in Wide-Area Networks. *Proceedings of IEEE SAINT 2001*, Jan. 2001.

[37] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proceedings of USENIX OSDI 2002*, Dec. 2002.