Virtual Services

A New Abstraction for Server Consolidation *

John Reumann†, Ashish Mehra‡, Kang G. Shin†, and Dilip Kandlur‡

†*Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122*

‡*Server Networking and Network Services, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598*

Abstract

Modern server operating systems (OS's) do not address the issue of interference between competing applications. This deficiency is a major road-block for Internet and Application Service Providers who want to multiplex server resources among their business clients. To insulate applications from each other, we introduce *Virtual Services* (VSs). Besides providing per-service resource budgets, VSs drastically reduce cross-service interference in the presence of shared backend services, such as databases and name services.

VSs provide dynamic per-service resource partitioning and management in a manner completely transparent to applications. To accomplish this goal, we introduce a kernel-based work classification mechanism called *gates*. Gates track work that propagates from one service to another and are configured by the system administrator via simple rules. They automate the binding of processes and sockets to VSs, and ensure that any work done on behalf of a VS, even if it is done by shared services, is charged to the resource budget of the VS that requested it. Using our experimental Linux 2.0.36-based implementation we applied them effectively to co-hosted Web servers. Thus, nearly eliminating performance interference between the co-hosted sites.

1 Introduction

It is becoming increasingly common and desirable for companies to outsource applications and services to Internet or Application Service Providers (ASPs) to reduce hardware and administration costs. ASPs save cost by sharing hardware, software licenses, and personnel among business clients. To minimize the number of system administrators required to run the ASP's site and to decrease the rate of system failure, ASPs invest in highly reliable and powerful servers. Since such server setups are not cheap, its resources should be highly utilized by sharing them among as many business clients as possible. In addition to the sharing of hardware resources, software resources may be shared across services. For instance, the DNS server will generally be shared across services and even business clients. While reducing the ASP's cost, aggressive sharing makes ensuring the Quality-of-Service (QoS) for the outsourced services difficult. Since ASPs must fulfill QoS contracts, a.k.a. service level agreements [22], they must tackle the performance interferences that result from the sharing of resources and services without deploying highly under-utilized and hence, cost-inefficient servers.

Recently, resource sharing has been addressed by the virtualization of resources in off-the-shelf OS's (e.g., *virtual Web hosting* and *virtual servers*) [2, 4, 6, 7, 8, 14, 21]. The essence of these concepts is that one physical server is split into several virtual hosts (VHs). Ideally, neither the client nor the server application is aware of the fact that it is executing on a VH and not on a real host. Initial implementations of this idea were content-based VHs. Here, a server would serve different content, depending on the IP address that was used to contact it, e.g., Apache's `VirtualHost` directive [14]. The performance interference that occurs between co-located VHs was not considered. To solve this problem, resource bindings for VHs were introduced [2, 4, 7, 20, 21]. With resource bindings, demand surges on one VH will no longer impact the performance of other co-hosted VHs. A service that is executed on one VH behaves as if it were executed on its own physical server. This still does not address the performance interference between services that may result from accessing the same backend service.

*The work reported in this paper was supported in part by the IBM Graduate Fellowship Program and by the NSF under Grant No. EIA-9806280.

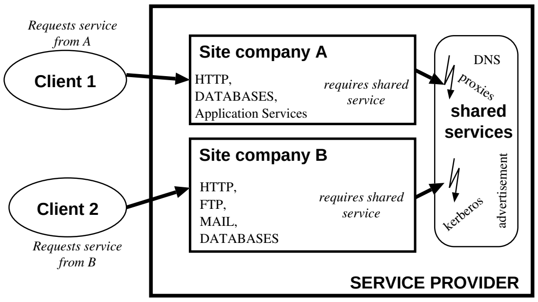


Figure 1: **Service-sharing destroys insulation**

Looking at the services that are hosted on a VH, we found that they vary between monolithic, which implement all service functionality themselves, and lightweight, which relay most of their work to other services. Therefore, the abstraction of a service does not directly coincide with the process boundaries imposed by the OS. Nevertheless, we define a *service* as the set of processes, sockets, and file descriptors that implement the service interface and share one address space. All other service activities (henceforth called just activities) that a service may trigger outside its own address space are referred to as *sub-services*. Since service providers often host similar services for different business clients, subservices often shared among them.

When services are shared among different business clients, VH-based insulation approaches fail (see Figure 1). Shared services like DNS, proxy cache services, time services, distributed file systems, and shared databases, are quite common. With shared services, an obvious question arises as to which VH should host these shared services. Since these services work on behalf of many other services, their resource bindings should be dynamic to reflect the *works-for* relation. The problem could be fixed by replicating shared services on each VH. However, in this case the consistency of individual shared services becomes a major concern, as does the inefficiency that results when hosting two identical services to maximize performance insulation.

To eliminate the performance interference caused by shared services, we introduce the notion of a *Virtual Service* (VS). VSs are an OS abstraction that provides per-service resource partitioning and management by dynamically binding service activities in a manner that is completely transparent to applications. Once an activity is classified as belonging to some VS, this VS association is maintained, regardless of the process context in which the activity continues. This means that the resource bindings for shared services are delayed until it is known who they work for. This automatic and delayed resource binding makes insulation between services possible, in spite of shared sub-services.

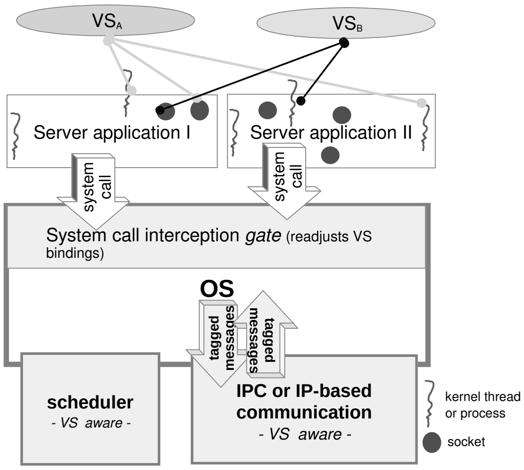


Figure 2: **Virtual Service Architecture**

In the VS architecture, the dynamic binding of activities to VSs is inferred by intercepting system calls within the OS using *classification gates* and analyzing the information that is passed to the call (see Figure 2). Gates track work that propagates from one service to another and are configured by the system administrator via simple rules. They automate the binding of resources (e.g., newly-created processes and sockets) to VSs, and keep track of any work done on behalf of a VS. Administrators can specify which system calls should be intercepted. Furthermore, they set up the rules that specify how the association between resources (e.g., processes, sockets, etc.) and VSs is affected by the intercepted calls. For example, a rule like: "If process P_1 `accepts` a service request from VS_x, the resulting P_1 activity should be charged to VS_x" can be configured easily. Our design also includes a VS-aware scheduler and packet dispatcher that enforce per-VS CPU and network shares.

The combination of VSs and gates permits resource management for dynamic communicating services without requiring any changes to the hosted services. While being transparent to the application, the system administrator must know the mechanisms by which work is relayed among VSs. Fortunately, this information can be inferred without application modification by system-call-tracing utilities or packet sniffers.

Other approaches that permit changing the resource bindings [2, 4] do not consider the problem of shared sub-services that do not readjust resource bindings on their own. Since the applications that the ASP hosts are usually developed for standard OS's, they are not resource-binding-aware. As Figure 2 shows, our transparent VS architecture brings resource management to such applications.

The following points summarize the key features of our

VS architecture:

- Dynamic resource bindings for shared services
- Application transparent resource management
- Applications may use several OS mechanisms to relay work to each other
- Minimal interference between VSs
- Modular implementation permits trade-off between the quality of insulation and the overhead incurred by the VS abstraction

The idea of a VS can be directly applied to distributed server farms within one administrative domain and shared VS name space. Even though collaborating services may be scattered over several machines, they still relay service requests to one another via system calls which the classification gates can intercept.

We summarize related work in Section 2. Section 3 details the design of VS and the dynamic management of VS associations. Section 4 describes our Linux-based implementation. Section 5 presents experimental results and quantitatively shows that the VS abstraction solves the problems faced by ASPs that co-host services. Section 6 summarizes our findings and future research directions.

2 Related Work

There are two approaches to server management: *resource-oriented* and *service-oriented*. Resource-oriented approaches like Resource Containers [2], Eclipse [3, 4], Capacity Reserves [15], and Hierarchical Scheduler [9] provide necessary low-level support for the partitioning of resources. Furthermore, they support relatively static bindings of resource consumers to these partitions. VS (proposed in this paper) and Workload Manager [1] are service-oriented. They charge services for their resource usage instead of creating static resource partition bindings for entities like processes, users, or sockets. Table 1 characterizes the properties of related approaches. This figure also highlights the novel features of VS.

Resource Containers (RC) is the most recent representative of resource-oriented server management. Any OS or application activity executes in the context of an RC. The RC may contain basic CPU and network shares and various count limits on the number of resource consumers that can be bound to it. To control application performance, processes must explicitly bind to an RC. Subsequent activities are charged to the associated RC, and resource limits specified therein are enforced. Unlike VS, the RC abstraction does not automate the binding of resource consumers to RC's.

The Solaris Resource Manager [20] is based on a resource reservation concept (called `lnodes`) which is equivalent to RC. In addition to the resource-reservation abstraction, `lnodes` are tagged with Unix user-group affiliations so that the resource context can be inferred from the user-group setting of current application activity. This mechanism reduces the need for manual resource bindings. The idea is to give each user-ID its own machine. Unfortunately, this concept fails if shared services do not change their user-ID when they work on behalf of different users. For example, the system's DNS server does not change its user-ID to that of the process requesting address resolution.

In the context of Eclipse's hierarchical reservation domains [4] Bruno *et al.* discuss in a recent paper [3] how Eclipse tackles the problem of sharing specific OS entities such as sockets among concurrent applications. Interference can be reduced by tagging each request that utilizes a shared resource with the appropriate reservation domain, thus delaying the resource binding of the shared OS abstraction. Request tagging is also used by VS. Unlike VS, Eclipse does not infer the tag for a request in the absence of application support and does not exploit these for the scheduling of an application that picks up a tagged request. Precursors of this work are the Hierarchical Scheduler (HS) [9] with configurable CPU scheduling policies and the Nemesis OS [10]. Nemesis provides comprehensive inter-application isolation for memory and file system. Both HS and Nemesis require applications to manage their own resource bindings.

Workload Manager's (WLM's) [1] notion of a *service class* is similar to the notion of a VS. Since WLM manages requests separately according to their *service class*, service sharing does not necessarily cause interference. Nevertheless, classifying requests into service classes is the hard part. For this purpose, IBM modified OS/390's services to classify all requests into service classes. This approach does not work for ASPs since they cannot modify hosted applications. Therefore, VSs provide a transparent work classification mechanism.

Scout [18] takes a somewhat different route since it is primarily designed to be used in embedded multimedia server designs. Scout's path abstraction tracks the flow of work across different OS layers. Resources are reserved on a per-path basis. Since paths are compiled into the kernel, resource consumption scenarios cannot change dynamically. For every new resource consumption scenario (i.e., new applications) the Scout kernel must be recompiled. In contrast, VSs can be configured dynamically to handle new scenarios of resource con-

	Dynamic System Domains	Eclipse BSD	HS	RC	Resource Manager	Scout	VS	WLM
OS	Solaris	Eclipse	Solaris	Digital Unix	Solaris	Scout	Linux	OS/390
Focus	VH on multi-processor	VH	Mulimedia end-host	abstract VH	per-user VH	Multimedia end-host	per-service resources	workload mgmt
Single server	Y	Y	Y	Y	Y	Y	Y	Y
Server farm	Multi-processor	N	N	N (pot. Y)	Multi-processor	N	Y	Y
CPU control	Y #CPU's/VH	Y	Y	Y	Y	program-mable	Y	priorities
Net control	Y (coarse)	Y	N	indirect	N	program-mable	Y	N
Disk control	Y (coarse)	Y	N	N	N	program-mable	N	Y
Memory control	Y (coarse)	N	N	N	Y	program-mable	Not yet	Y
Inference of resource principal	VH	N	N	some TCP-based	Unix user/group	N	rule-based	app-based
Recognize work delegation	N	N	N	N	N	hardwired into OS	Y intercept syscall	N
Application changes	Y	N	Y	Y	Y	OS part of application	N	Y

Table 1: **Resource- and service-oriented server management solutions**

sumption and service interaction.

Sun's Dynamic Enterprise 1000 [21], Solaris Resource Manager [20], and Ensim's recent VH product ServerXchange [7] are noteworthy commercial VH implementations resembling Eclipse. Other popular commercial solutions, such as Cisco's LocalDirector [6], HydraWeb [11], and F5's BigIP [8] are geared towards increasing the capacity available to ASPs through load-sharing in server clusters. These solutions also provide some coarse insulation between co-hosted services by shaping request flows. This mechanism applies only to well-known services (e.g. Web servers) since requests must be parsed by a load-managing frontend. Hence, insulation fails when ASPs co-host proprietary services and when the workload created by individual requests differs significantly among co-hosted sites.

3 The Virtual Service Abstraction

The VS abstraction treats services that utilize sub-services as if they were a single application executing on its own dedicated server. To create this illusion, a VS is associated with a basic resource context (Figure 3). The resource context summarizes the resource limits and statistics for activities that execute on behalf of the VS (Section 3.2). Figure 3 also shows typical VS members. Section 3.3 discusses how to track this dynamic member set if applications do not manage VS-membership on their own. In Section 3.4 we discuss the seamless integration of the tracking of VS-membership and resource limit enforcement into a *gate* abstraction. A gate filters entry and exit of system calls that are relevant with respect to VS-membership changes. Section 3.5 explains the gate's response to resource limit violations.

3.1 System Model

Our approach uses tags OS entities, such as processes, sockets, IPC shared memory segments, etc., with VS information. All modern OS's have these entities and already tag them with other information. We also have to assume that all requests for service are received via explicit system calls, such as communication sockets, IPC shared memory segments, message queues or pipes. This restriction is due to the fact that automatic tracking of changes in VS-membership depends on being able to intercept the interaction between cooperating services.

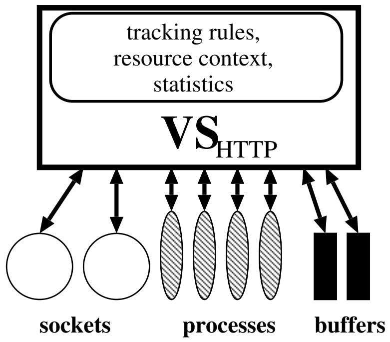


Figure 3: **A VS and its members**

This means that VSs cannot manage services that hide their relaying of work. For example, if two services use some VS-unaware server as a message relay, it is impossible to infer their cooperation. User-level thread libraries are another form of hidden communication since the application's switching between different requests is hidden from the OS. Here, the thread library must rebind the process to the correct VS every time a thread-switch occurs.

3.2 Setting up a Virtual Service

Each VS is uniquely identified by its descriptor (Figure 4). To allow the system administrator to manage VSs, each service has an integer virtual service identifier (VSID). The VSID is guaranteed to be unique on each machine. For VSs that use resources on multiple machines, we leave it up to user-space software to guarantee the uniqueness of identifiers. When setting up the distributed service, external administration software can force a specific VSID onto the newly-created VS. Such global VSID's are taken from their own number range and will never conflict with local VSID's.

Like RC's [2] we advocate hierarchically-nested resource contexts. Hierarchy is necessary because an ASP's clients should be able to decide themselves whether they want all of their services to share resources or whether they want to insulate them from each other.

The parent field of the VS structure points to the parent VS. Oftentimes parent VSs will be used to implement abstract VSs, i.e., placeholder services to which all services of an ASP's business client belong. The highest-level service is the `root_service` with VSID 0, which accounts for all unclassified work.

Hierarchy is again reflected in VS attributes such as resource usage statistics and resource limits. By default, newly-created VSs share the attributes, i.e., resource control settings and statistics (CPU time used, packets sent, etc.) of their parents. This means that the fields in the child refer to its parent's pendants (Figure 4). To manage the child service directly, attributes of interest need to be detached from the parent. For example, to control the number of processes for a sub-service, one detaches the sub-service's process count limit via the `servctl(DETACH_PROCESS_LIMIT)` call.

To instantiate a cluster-wide VS, the administration software must create VS descriptors with one global VSID on all cluster nodes. On each of those nodes, local resources may be reserved using the VS descriptor's resource context. Before reserving VS resources, the administration software will monitor the VS's resource consumption via the statistical VS attributes. Once enough statistics are available, resource reservations will be calculated to stabilize VS performance. This calculation is difficult and requires a full-fledged monitoring-feedback algorithm, which is outside the scope of this paper. We experienced that cluster-wide VS management using a unified VSID name space simplifies the implementation of such an algorithm.

Most of the VS state discussed so far could potentially be realized using RC's [2] or Reservation Domains [4]. However, they do not provide configurable classification rules. Classification rules indicate how VS-membership is to be updated when certain system calls are invoked by specific VSs. For instance, if a process member of the VS in Figure 4 calls `fork`, the OS knows exactly that this is a way of relaying work and that the created process should inherit the parent's VS affiliation.

3.3 Tracking Service Membership

There are two ways of tracking the members of a service: either they are announced or the OS infers who they are. In our VS framework, membership is mostly tracked by the OS without requiring continuous application or administrator intervention. Nevertheless, especially at service startup time it can be efficient to create some associations between VSs and other OS entities explicitly. For example, if one knows that one specific kind of service request (identified by its own VS) always enters the system through one specific process or socket, a manual binding of these processes or sockets to the VS should be used. This avoids having the OS infer repeatedly that these entities and the VS belong together.

As a VS begins to respond to requests, new sockets, processes, and IPC resources may be created. Each of them must be associated with a VS because they incur system load and are used to relay work. Usually these new members are not added explicitly since the administrator does not even know of their existence and the applica-

VSID	resource limits	resource statistics	classification rules	rule priority	parent
global 123	CPU rate comm rate ...	actual CPU use actual net use ...	fork-ed children map to VSID 123(error EAGAIN) connect-ed service maps to VSID 123(error none)	1	global 120

may refer to parent attribute

Figure 4: **The VS descriptor**

tion does not cooperate with the VS abstraction. Therefore, membership for these new entities is implicitly determined by the classification rules in the creator's VS descriptor.

Not only do new entities need to be associated with a VS, but VS-membership may also change over time. For instance, if some process is observed to be operating on a particular data set that is characteristic of some separately-managed VS, the process is added to that VS and removed from its current VS.

Classification, i.e., associating an OS entity with a VS, takes place when the OS can infer something about the application, i.e., at system call time. We can limit VS-membership inference to those times because we assumed in Section 3.1 that VSs interact with each other over a limited set of OS mechanisms. This means that the *works-for* relation cannot change unless a system call is invoked. Therefore, there is no need to update VS-membership at any other time.

The **classification rules** that the OS examines at system call interception consist of a conditional clause, which defines when the classification rule is applicable, and a classification directive. This is formalized as:

$$(\,\texttt{syscall}, S_1, \ldots, S_m, P_1, \ldots, P_n) \rightarrow (S'_1, \ldots, S'_m)$$

Here S_i represents the VS of the i-th affected entity. For example, the only affected entity in the `exec` call would be the calling process. The calling process's VS is always identified by S_1. P_j represents the j-th intercepted property, for example, the program name passed to `exec` or the incoming IP address of an `accepted` connection. Properties are not OS entities. A classification rule also specifies S'_i: the resulting VS classification of the i-th affected entity. This classification is applied only if the conditional (left-hand side of the rule) matches the intercepted system call. S_i and P_j may be wildcards. Our prototype implementation requires S_1 to be specific. The system call is always specific, since the dimensionality of the condition tuple depends on it. One way of managing the rules would be to store them in a system-wide table, which would ease the integration of VSs into RC's or Eclipse. Our implementation reduces lookup times by storing rules solely inside the VS descriptor that matches S_1.

Conflicting rules: Rule matching can lead to ambiguity. Multiple condition tuples may match the current system call interception. To solve this problem, VSs are prioritized. The rule that matches the highest-priority VS explicitly is used to determine the resulting classification. Should there be a tie between several rules, the most specific rule is applied. If this does not resolve ambiguity, the result is unspecified.

In the remainder of this subsection, we will discuss the different types of classification triggers. Table 2 casts common UNIX system calls into these categories.

Creation: If an entity, A, creates another OS entity, B, B's future VS affiliation depends solely on A's VS affiliation. Examples are the creation of sockets, IPC shared memory segments, message queues, pipes, and the like. The canonical default rule is for the created entity to inherit its creator's VS affiliation.

Communication: Communication is used to relay work within and beyond machine boundaries. Therefore, intercepting intra-VS communication is essential to VS maintenance in server farms. If it is possible to determine the VS affiliation of each request that is picked up by a service, the resulting activity can be charged to the correct VS. This does not depend on whether the request originated locally or remotely.

Communication affects at least three entities: *sender*, *receiver*, and the *message* itself. To make inter-process communication more efficient, most OS's implement asynchronous communication as opposed to the *rendezvous* concept. This adds sockets, pipes, and the like to the set of affected entities, each of which may be reclassified upon system call interception.

Due to the temporal separation between sending and receiving a message, pausing to reclassify the affected entities is difficult. Therefore, communication-based VS tracking is done in two stages. First, when the message leaves the sender it is tagged with a VS affiliation, much like what is done in the case of creation-type calls. This can be skipped if the communication is a one-to-one connection. In this case, the connection itself is labeled at setup time with a VS affiliation that implicitly applies to each message that passes through it. The second stage is message consumption. At this time, the receiver's VS affiliation may change based on its previous

<table>
<tr><th>Category</th><th>System call</th><th>Proposed classification mechanism</th><th>Proposed VS error</th><th>Used by</th></tr>
<tr><td rowspan="5">creation</td><td>fork</td><td>Classify new process based on forker's VS</td><td>EAGAIN, block</td><td>Multi-threaded services</td></tr>
<tr><td>open</td><td rowspan="3">Tag created file descriptors with a service affiliation based on creator's VS</td><td rowspan="3">block</td><td>User services, FCGI</td></tr>
<tr><td>socket</td><td>All network services</td></tr>
<tr><td>pipe</td><td rowspan="2">User services, FCGI</td></tr>
<tr><td>shmctl</td><td>Tag new shared memory segment based on creator's VS</td><td>EINVAL, block</td></tr>
<tr><td rowspan="11">communication</td><td>accept</td><td>Classify caller based on its VS, the VS of the incoming connection and incoming source address</td><td>EWOULDBLOCK, block</td><td>User services, FCGI</td></tr>
<tr><td>connect</td><td>Classify caller and connecting socket based on the destination IP + port</td><td>EINPROGRESS, block</td><td>Frontend services, HTTPD with distributed FCGI</td></tr>
<tr><td>send</td><td rowspan="3">Classify caller and sending socket based on destination IP + port</td><td rowspan="6">EWOULDBLOCK, block</td><td rowspan="3">Frontend</td></tr>
<tr><td>sendmsg</td></tr>
<tr><td>sendto</td></tr>
<tr><td>recvmsg</td><td rowspan="3">Classify caller and receiving socket based on incoming packet's VS and IP + port</td><td rowspan="3">NFS, DNS, RPC, Multimedia</td></tr>
<tr><td>recvfrom</td></tr>
<tr><td>recv</td></tr>
<tr><td>shmat</td><td>Classify caller based on shared memory's VS</td><td>EACCESS, block</td><td>Apache (thread synch)</td></tr>
<tr><td>msgsnd</td><td>Tag message based on caller's VS</td><td>block</td><td rowspan="2">Proprietary services</td></tr>
<tr><td>msgrecv</td><td>Classify caller based on the incoming message's VS</td><td>ENOMSG, block</td></tr>
<tr><td>synchronization</td><td>semop</td><td>Classify caller based on the semaphore's VS</td><td>block</td><td>Proprietary services</td></tr>
<tr><td rowspan="3">transformation</td><td>exec</td><td>Classify caller based on the executed program's name</td><td rowspan="3">block</td><td>HTTP-CGI, Inetd, rexec, rsh</td></tr>
<tr><td>listen</td><td>Classify caller based on its and socket's VS</td><td>TCP + Unix Domain socket-based services (Fast CGI)</td></tr>
<tr><td>bind</td><td>Classify caller and socket based on the IP + port pair</td><td>Standard services</td></tr>
<tr><td rowspan="2">communication, synchronization, transformation</td><td>write</td><td>Tag the message based on caller's VS or inherit file descriptor VS</td><td rowspan="2">EAGAIN, block</td><td rowspan="2">All services</td></tr>
<tr><td>read</td><td>Classify caller based on read message's VS or inherit file descriptor VS</td></tr>
</table>

Table 2: **System calls that affect VS-membership**

VS affiliation and the received message's VS affiliation.

Synchronization: The set of affected entities in synchronization includes the executing process(es) and all process(es) in the wait queue for the synchronization primitive. Activities that are performed under the protection of a synchronization primitive may be associated with its VS.

Synchronization can also be used to infer collaboration among a set of processes. Previously-unclassified processes may inherit the VS affiliation of the synchronization primitive. This is an effective tool since many multi-threaded server applications expose their process sets when they synchronize for thread control purposes.

The process(es) that execute under the protection of the synchronization primitive must not stall processes in the wait queue because otherwise, priority inversion [13] will result. This is also a problem when a single-threaded sub-service is shared among several VSs. This will be discussed further in Section 4 (`accept`).

Transformation: Whenever the kernel intercepts a characteristic argument to a system call, it is possible to classify the caller and other affected entities more accurately. For instance, the program name in the `exec`-call allows a more accurate VS classification of the active process if the program is typical of a specific VS. Other frequently-used system calls that may alter VS classifications without relaying work are `setgid`, `setpgrp`, and `setuid`.

3.4 Virtual Service Gates

Whenever a VS receives a new member, resource limits could potentially be violated. This means that classification and resource limit enforcement are inseparable. Therefore, we introduce *gates*, a combination of system call filtering and VS classification. Each system call that is used to track VS-membership is controlled by a gate.

If the gate's filtering code indicates a resource limit violation as a result of the new classification, the system call will either fail with an administrator specified `errno` code, block, or execute in best-effort mode. Otherwise, VS-membership is updated as specified in the classification rules. Figure 5 depicts the basic anatomy of a gate:

1. The *prefilter* checks whether the caller is (a) classified and (b) whether its VS affiliation permits the execution of the gated call.

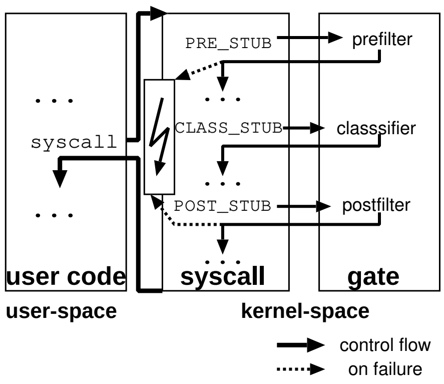


Figure 5: **Gated system calls**

2. The *classifier* applies a matching classification rule. To execute the classifier for creation-type calls it executes after the new resource has been created.

3. Finally, the *postfilter* checks whether the resulting classification violates any VS resource limits. The resource limits we considered are: count limits on the number of processes and sockets. Other resource limits, such as CPU and network bandwidth are enforced silently by the packet and CPU schedulers and need be checked by the gate mechanism. If a resource limit is violated, the system call fails or retries as is described in the next section.

3.5 Failing System Calls

Gates may detect resource limit violations. For example, during the execution of `fork` it may become apparent that successful call completion would result in a violation the VS's process count limit. The appropriate remedy is application-dependent. One may decide to:

1. Wait until VS resources become available.

2. Return an error to the caller indicating resource exhaustion.

3. Not apply the classification that led to the resource limit violation and silently reclassify the caller as best-effort. If the best-effort VS has exhausted its resource share, there is no other option but to fail the system call.

In the first case, the OS will add the caller to a FIFO wait queue for the requested resource. For example, in the case of `fork` this means that the forker will sleep until the VS's process count drops below the process count limit. The resulting delay may not be acceptable to the calling application.

Applications that cannot be delayed, should receive an error upon resource exhaustion. Unfortunately, existing applications may not be able to handle arbitrary errors. Therefore, we leave it up to the administrator to configure the error that will be raised if a VS-level resource limit violation is observed at a particular gate. In this way, only errors that the application is able to handle will be raised. For example, the administrator may choose to raise the `EAGAIN` error for some VS that exceeds its process count limit upon `fork`. This behavior is specified at the time of gate configuration [e.g., `servctl(SET_FORK_POLICY, VSID`$_x$`, ... , EAGAIN, ... )`]. Most server applications are capable of handling errors that result from resource exhaustion gracefully. They simply record the error in a server log-file to support system tuning. If neither blocking nor returning an error is acceptable to the hosted service, the execution should continue in best-effort mode.

4 Implementation

We added VSs as loadable modules to the 2.0.36 version of the Linux kernel; located at http://www.eecs.umich.edu/~reumann/vs.html. Figure 6 shows dependencies among the VS modules. To implement the gates, we added only a few lines of call-back code to the intercepted system calls to trigger VS classification. The VS structure itself (Figure 8) contains the previously-described membership information, statistics, and resource limits. The VS structure, VS hierarchy management and most of the gates are portable since they hardly depend on Linux internals. The placement of the call-back code in the original system calls is Linux-specific.

We implemented VS-level *fair-shares* [12, 15] for CPU and network to provide strict VS-level resource guarantees. VSs that are neither directly nor indirectly (via a parent VS) associated with a share are scheduled on a best-effort basis. Best-effort VSs use all unreserved resource slots. Any excess capacity is shared between VSs that own resource shares in a round-robin fashion (see also firm Capacity Reserves [16]). The implementation of VS resource shares is not portable across platforms. Nevertheless, numerous implementations of capacity reserves and fair-shares exist. Therefore, requiring VS-level fair-shares does not limit the applicability of our approach.

VS statistics are cumulative aggregates of the VS's members' statistics. The attributes include a wide range of statistics that Linux keeps for processes and sockets, such as page faults and virtual time elapsed.

To set up the VS hierarchy and adjust CPU limits,

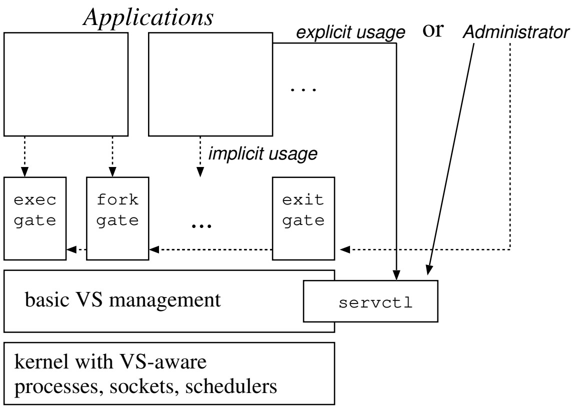


Figure 6: **VS module dependencies**

VS membership, policies, attribute inheritance, resource limits, and query VS attributes, the OS offers a new system call (`servctl`). It takes a command, the size of the argument, and an argument structure as parameters.

Gates are implemented as loadable modules. We currently support `fork`, `exit`, `exec`, `open`, `accept`, and `socket` gates. Upon insertion of a gate module, the call-back stubs that are placed in their corresponding system calls are activated so that the gate's *prefilter*, *postfilter*, and *classifier* are executed each time the control flow of a server application passes through the intercepted system calls. Each gate also registers its own `servctl`-handler to enable gate configuration.

The advantage of our modular gate design is that one only needs to add those gates to the kernel that are absolutely necessary to classify VS membership and insulate services. This is very important because the insertion of each gate into a running kernel increases system overhead (see Section 5 for more detail). The remainder of this section describes our implemented gates.

Fork: Upon interception of `fork` the created process is classified as a new member of some VS. To determine the resulting VS affiliation, we check the `fork_policy` object of the creator's VS. The `map_to` attribute of the `fork_policy` specifies the affiliation of the created process. If the VS specified by `map_to` has reached its process count limit (set via the `servctl` call), the failure behavior that was configured for that VS is invoked (Section 3.5). Figure 7 shows a high-level control flow graph for this gated system call.

Exit: If a process exits — including ungraceful `SIGSEGV` and other uncaught signal exits — it must be removed from the VS with which it is associated. This gate is not configurable.

Exec: Upon calling one of the `exec`-family system calls, the caller can be reclassified based on the name of the program that was invoked. The gate code checks the name of the program against a hashed mapping table, i.e., the `exec_policy` field in Figure 8.

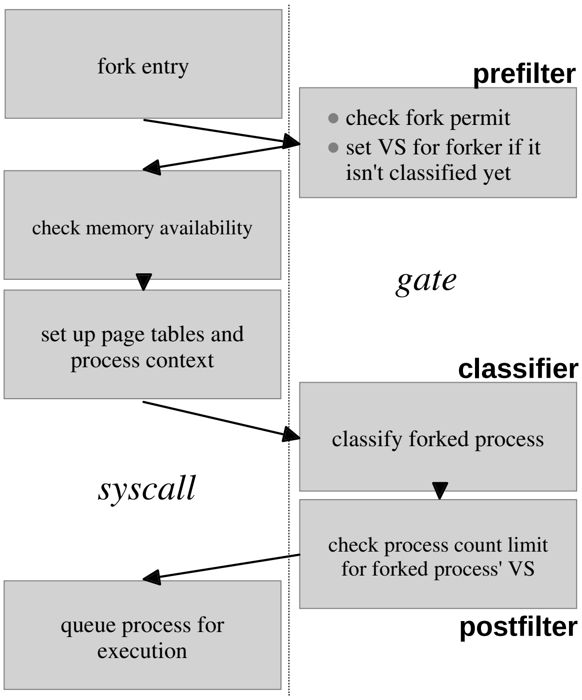


Figure 7: **Control-flow of the fork gate**

Open: The `open` gate acts like the `exec` gate. The only difference is that the file descriptor may be tagged with a VS affiliation at the same time. Moreover, the `open` gate uses a prefix-tree to match the file names. Thus, whole directories — identified by a shared prefix — can map to one VS. This is important because large numbers of data files residing in one directory subtree may yield identical VS classifications.

Socket: The socket gate resembles the `fork` gate. The `socket_policy` of a VS specifies the future VS affiliation of the created socket. Once messages are relayed via such a classified socket, they are tagged with the VS affiliation of the socket in their IP Type-of-Service field (TOS), thus allowing VS information to propagate over the network. Since the TOS field may be used by DiffServ to provide differential QoS in a WAN, this field can only be used inside server clusters. If the TOS field cannot be used or one needs more than 256 VSID's (the TOS is eight bits wide), one may introduce a new IP-option [17] to hold the VSID. This option should be set in every fragment of the IP datagram to facilitate VS-aware routing.

Close: Closed file descriptors' and sockets' VS affiliation must be removed.

Accept: The `accept` gate is quite complex. It first de-

```
struct service_struct {

            int sid;
            struct service_struct *parent;
            char name[MAX_SERVICE_NAME_LEN];
            int precedence;

            // int_or_ptr is either a value or a
            // pointer to the parent's int_or_ptr
            int_or_ptr process_count;
            int_or_ptr socket_count;
            int_or_ptr byte_count;
            int_or_ptr vtime;
            int_or_ptr majflt;
            int_or_ptr minflt;

            member_struct *processes;
            member_struct *sockets;
            member_struct *services;
            fork_policy_struct fork_policy;
            exec_policy_struct exec_policy; ... more ...
            cpu_policy_struct cpu_policy;
            comm_policy_struct comm_policy;
};
```

Figure 8: **The VS struct**

termines the highest-priority VS among the caller, listening socket, and incoming connection. Then the winning VS structure is checked for a VS mapping based on the incoming IP address and the VS affiliation of the listen-socket, process, and incoming connection. The VS affiliation of the incoming connection can only be determined if it was initiated by another server with our VS support and its VSID is from the global VSID range. The VSID is stored in the incoming SYN packet's IP TOS bits. For local `accept`s, the VS of the incoming connection is the connecting socket's VS affiliation. Both socket and receiver may be reclassified.

The difficulty with `accept` is that it should not block if the first pending connection on the listen queue leads to a violation of resource limits. There may be a connection that can be `accept`ed without violating any VS resource limits. Therefore, our implementation scans the listen queue for the incoming connection whose VS has utilized its resource reservation the least.

Concurrent Gate Versions: A powerful feature of our implementation is to allow multiple versions of a gate to be loaded at the same time. Hence, VSs may specify which gate version they want to use when their process members invoke the corresponding gated system call. This way it is possible to eliminate unnecessary checks for specific VSs. For example, if `fork`-ed off processes should always inherit their parents' VS affiliation, it is unnecessary to check for a (`fork`, VSID$_x$) $\rightarrow$ VSID$_x$ mapping as is required for general VS classification. One can implement one `fork` gate version that always applies the parent's VS affiliation to the `fork`ed child. Another example is the `accept` gate, which is quite complex in its general form. In a server-farm setup it is likely that incoming service requests are already classified by the frontends and that the applications that process requests in the backends only need to inherit these classifications. This reduces the complexity of the backends' `accept` gates. We used such an optimized version of the accept gate in our experiments. In our experiments incoming requests were classified as they were picked up by the HTTP server. Whenever the HTTP server relayed work to a shared backend Fast-CGI (FCGI) service, the backend FCGI inherited the classification of the requesting HTTP server process.

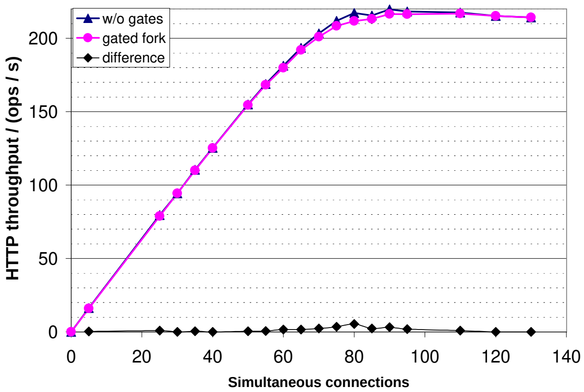


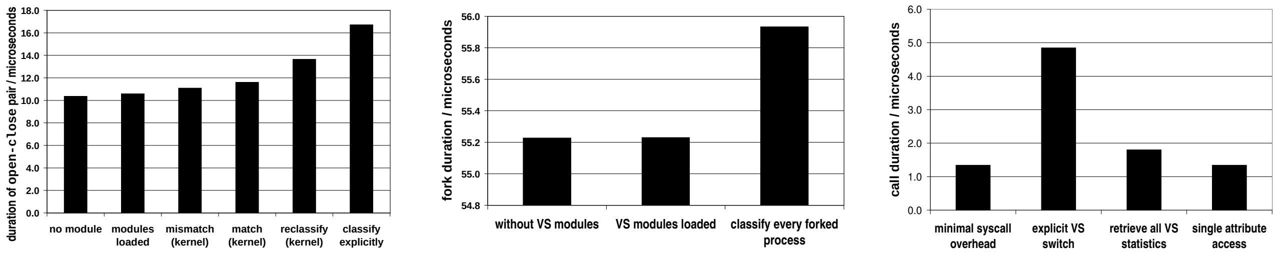
Figure 9: **VS effect on HTTP throughput**

5 Evaluation

We evaluated the performance of the VS architecture on a small Web server running on a Dell 450 MHz Intel Pentium II PC with 448 MB RAM and one UDMA HDD. The clients, three 300 MHz Pentium II machines with 128 MB RAM each, were connected through 100Mbps Ethernet. We measured the performance of Apache 1.3.6 (HTTP 1.1) on this platform running on the Linux 2.0.36 kernel. The workload was generated by the commercial SpecWeb99 benchmark [19]. SpecWeb99 attempts to model a realistic workload including 30% dynamic requests. The size of the file sets grows linearly with the number of simultaneous connections offered to the Web site. Therefore, it generally does not completely fit into the server's file cache. Dynamic requests and the use of Apache explain the low HTTP throughput of the server (ca. 220 ops/s). Since the VS abstraction is an application-transparent mechanism, neither applications nor libraries had to be modified. The management of the VS hierarchy and gate configuration was done from the command line using utilities that feed their arguments into the appropriate `servctl` call.

5.1 Baseline Performance

Basic performance measurements show that the dynamic VS classification layer degrades overall system performance only minimally. If we intercept a complex sys-



(a) `open/close` loop (b) `fork` performance (c) VS management performance

Figure 10: **Performance of intercepted and new system calls**

tem call like `fork`, the overhead of classifying the new process is small — only 1.3% — (see Figure 10(b), classify). Nevertheless, the raw performance of intercepted system calls can decrease significantly if the intercepted call is very simple like `open`. We observed 30% cost increase for the `open/close` pair if the VS affiliation changed with every execution of the loop (reclassify) in Figure 10(a). Just finding a classification rule (match) or not finding one (mismatch) is much cheaper. The high relative overhead for simple calls results from the almost constant classification overhead. An important point shown in Figure 10(a) is that binding processes to VSs from user-space (explicit classification) performs much worse than kernel-based classification because of the system call overhead. Explicit classification requires the executing process to classify itself and the resources that it uses and creates by calling the `servctl` system call. In our measurements we compared the performance of a sample program using (implicit) classification rules against a modified version of the program that explicitly updated affected VS bindings. The performance numbers strongly support the use of kernel-based (implicit) classification. For the sake of completeness, Figure 10(c) summarizes the cost of querying VS attributes and administering the VS hierarchy.

To estimate the overall performance impact of the kernel modifications including scheduler changes, resource limit enforcement, and the cost of system call interception, we measured how the performance of the Apache HTTP server [14] is affected by the OS changes. Figure 9 shows that the VS abstraction affects the system's HTTP performance only up to 2.5%, depending on the number of simultaneous client connections. The bimodal shape of the performance loss graph in Figure 9 can be explained as follows. Apache keeps some spare processes alive to serve incoming connections faster. Once the number of simultaneous connections offered to Apache increases to an extent that there are not always enough of these spares, Apache begins to fork more connection-handling processes on-demand, which explains the increase in overhead up to 80 simultaneous connections. Beyond this point, the file system cache hit ratio goes down so that the low performance of the file system begins to dominate overall system performance, thus decreasing the relative impact of our OS changes.

5.2 Implementing VHs using VSs

Another series of experiments on Apache shows that the VS abstraction may be used to insulate VHs. Unlike other applications, Apache itself provides some basic resource controls (process count limits) to insulate VHs. We studied the insulation properties that Apache can provide in comparison with those of VSs. The goal was to divide the previously-measured server into two VHs of equal capacity.

In the experiments, two copies of Apache were executed on the same host, each listening on its own IP address using IP aliasing for the Ethernet interface. Running two copies of Apache, each instance can be controlled by adjusting the `MaxClients` directive, which limits the number of concurrent sessions for each site. This is an effective means of performance insulation if the average work per HTTP operation is known for each site. Since both sites have their own copy of similar content, we could achieve a good division of resources by setting the `MaxClient` directive for the Apache servers to the same value.

To test VS-based insulation, the Apache servers were launched as if they were run on their own physical hosts (using very large process limits). We created two VSs, `www1` and `www2`, for which we specified the `fork` classification rules:

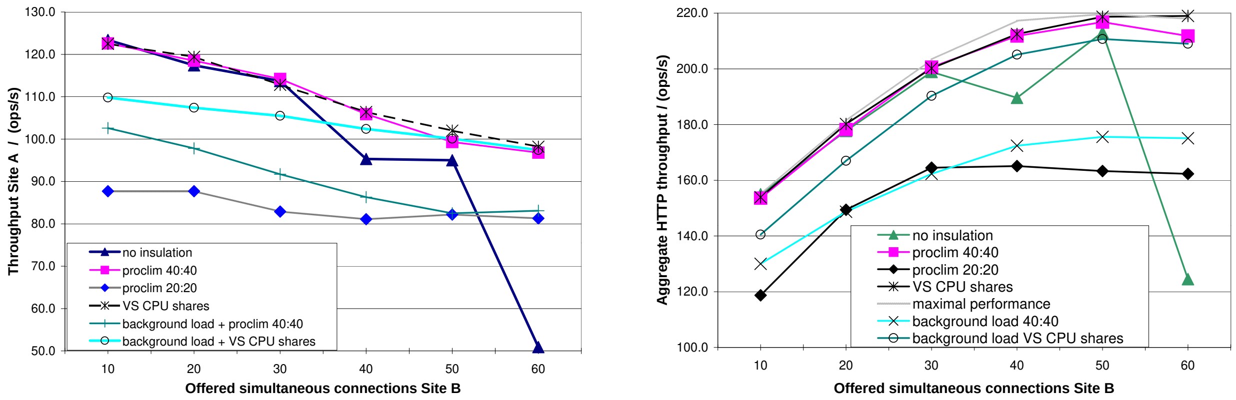
$$(\texttt{fork}, \texttt{www[1|2]}) \rightarrow (\texttt{www[1|2]})$$

Each site's initial `httpd` process was explicitly added to its corresponding VS via a simple command line utility:

```
$> svcaddprocess <VSID> <PID>
```

Each site was given a 50% CPU share.

In the measurements that are discussed here, Site A was offered a constant load of 40 simultaneous connections



(a) Site A performance in a two-site setup

(b) Aggregated performance in a two-site setup

Figure 11: **Performance loss when hosting two sites of equal capacity on one server**

while Site B was offered between 10 and 60 simultaneous connections. We chose these parameters because the server saturates — diminishing HTTP throughput gains — when offered 80 simultaneous connections.

Without insulation between the sites, A's performance degrades significantly once the server is offered a total of 70 simultaneous connections (A=40, B=30) [see Figure 11(a)]. From this point on, B begins to steal resources from A, thus contaminating the file cache to A's disadvantage. The lack of insulation can be fixed in Apache itself by restricting the maximal number of concurrent processes. This comes at the expense of some loss of aggregated performance under peak load [Figure 11(b)]. This loss is due to the fact that incoming requests must be rejected when the process limit is reached. This queuing phenomenon — for M/M/m/c systems described by the Erlang-loss formulas [23] — is especially evident when looking at the smaller process count limit (20:20). VS CPU shares eliminate this problem.

Apache's process limits also fail when background activities compete for CPU time, e.g., monitoring. To simulate the effects of background load, ten background load generators were invoked. As expected, aggregated performance and A's performance drop significantly if Apache's process limits are used for site insulation. In contrast, the VS abstraction keeps A's performance stable since only non-dedicated resource slots (beyond A's and B's resource limits) are used to process background load. Therefore, VS-based insulation performs better than Apache's own support for VHs.

One may argue that a modified, CPU-share-aware Apache could achieve the same quality of insulation. VSs obviate the need for modifying applications to get a better handle on performance management.

Since this experiment did not involve access to any shared services and work is relayed only from a parent process to its child, Eclipse or RC's could probably be tuned to perform just as well as VSs. Beyond establishing the competitiveness of the VS approach, the next set of experiments focuses on its main contribution.

5.3 Insulation Despite Service Sharing

Instead of letting the sites A and B execute CGI scripts to serve the advertisement banners (part of the benchmark), a shared, single-threaded Fast-CGI server (FCGI) was used. Queuing theory suggests that the impact of this shared FCGI will be the worst when (a) it exhibits highly variant execution times and (b) a high percentage of requests are forwarded to it. Therefore, we modified the FCGI to execute a busy wait cycle randomly chosen between 0 and 10 ms (uniformly distributed), before serving incoming requests for advertisement. Furthermore, the percentage of advertisement banners requests was raised from 13% to 30% on each site. Other dynamic requests were eliminated from the benchmark's workload. The Apache sites (A and B) used a TCP connection to retrieve advertisement from the shared FCGI, which was located on the same machine as the two sites. Since we used a TCP connection, we could have also moved the FCGI to a remote server without breaking the VS paradigm. Nevertheless, the remote server would have been so underloaded that interference effects would not have been easy to observe. Moreover, the time required to invoke a remote FCGI would have severely limited overall HTTP throughput. The load offered to Site A was kept at 30 simultaneous connections while the load offered to Site B increased from 10 to 60 simul-

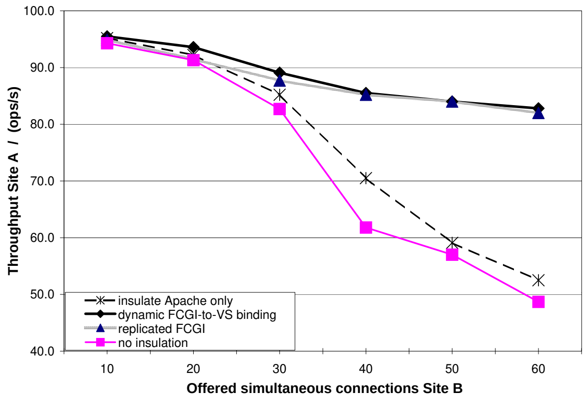


Figure 12: **A's performance while increasing load on B**

taneous connections — with the changed dynamic mix, the server became saturated at a total of 60 simultaneous connections of offered load.

As in the experiments of the last section, two VSs (`www1` and `www2`) were created and each was assigned half of the machine's CPU capacity. The first experiment (Apache insulation only) executes the FCGI outside the VS context of either site so that it could utilize all unused server capacity.

In the second round (dynamic FCGI-to-VS binding) we loaded the additional `accept` and `socket` gates to police access to the FCGI, which received its requests via TCP. The following classification rules were instantiated:

$(\texttt{accept}, \texttt{www[1|2]}, \text{req} = \texttt{www[2|1]}) \rightarrow (\texttt{www[2|1]})$

The `accept` rules cause the FCGI to change its resource binding if it is executing in the VS context of `www1` (`www2`) and receives a request from `www2` (`www1`) to `www2` (`www1`). Moreover, `accept` reorders requests in the order of their VS's remaining resource share as explained in Section 4. The default `socket` rule associates a new socket with its creator's VS. This ensures that requests sent to the shared FCGI will have appropriate TOS markings. To establish a baseline for optimal insulation, we replicated the FCGI script in each VS context (replicated FCGI). This cannot be done in real setups because many applications are not designed to be replicated.

As Figure 12 shows, sharing the FCGI without the `accept` gate breaks the insulation between sites A and B (Apache insulation only); the performance of Site A decreases rapidly as the load on Site B increases. This effect can be traced back to the contention for the shared FCGI. With the `accept` gate (dynamic FCGI-to-VS binding), the performance of Site A drops at a much slower, nearly the pace for replicated FCGI. The benefit of using the `accept` gate is a performance improvement for the well-behaved site (well-behaved means that its clients do not overload the site) of approximately 60% under maximal load. Further experiments show that the `accept` gate for dynamic VS bindings performs almost as well as if the shared service were replicated for each VS (replicated FCGI). The ill-behaved Site B suffers from overloading its CPU share. This results in a 10% loss of aggregated performance compared to the ideal case of a replicated FCGI under peak load. The reason for this is that the ill-behaved site uses its resources mainly on serving static HTTP requests. Only when the number of queued-up FCGI requests is large will its FCGI requests be processed. During those times Site B operates mostly sequentially.

Without changing Apache this problem could not have been solved using RC's or any other approach presented in Section 2, because the resource binding for the FCGI must be dynamic, assuming it cannot be replicated.

6 Conclusions

We demonstrated that VSs are an effective, application-transparent resource management abstraction when sub-services are shared across business clients. Furthermore, our implementation showed that VSs can be integrated into an off-the-shelf OS without incurring much additional overhead. To manage VSs, a limited understanding of the managed applications suffices. In particular, one needs to know how services and sub-services interact. On the basis of these knowledge, the VS architecture transparently and dynamically updates the VS binding for service activities and thereby their resource bindings.

VSs are shown to be able to emulate the VH abstraction. Furthermore, we have shown that VSs provide sound insulation between competing services in spite of shared sub-services. ASPs who multiplex hardware and software resources among their business clients, benefit greatly from the proposed solution. Given the great interest in the outsourcing market, future versions of commercial resource management approaches such as WLM or Sun Resource Manager, will consider the interference caused by shared services between otherwise well-insulated services. They may use VS tracking to minimize this interference, thus improving resource management for multi-tier services significantly.

Since the VS architecture is extensible, one may choose only a small set of classification mechanisms and limited configuration options for gates. This allows a staged integration of VS tracking into off-the-shelf OS's. VS can also be integrated easily by putting the classification rules into a separate look-up table. Then a VS descriptor reduces to a RC or Reservation Domain. Therefore, it is possible to augment RC's or Eclipse to provide VS-like dynamic resource bindings by introducing a classification table and intercepting the VS relevant system calls.

In spite of the overall acceptable performance of our experimental implementation, there is still sufficient room for improvement. To speed up classification in a commercial OS, filtering and classification should be tightly integrated into the intercepted system calls as opposed to simply placing call-back hooks inside system call code — calling an empty C function on a 300MHz AMD K6 already takes 9μs. A tighter integration would also avoid duplicate lookups of processes, file descriptors, etc., once to execute the system call and another time to execute classification.

The primary remaining issue in insulating co-hosted sites from each other using VSs is file cache management. To improve insulation, each disk-bound VS should be equipped with its own file cache [5]. To accomplish this goal, the `inodes` in the file cache must be tagged with their VS affiliation. Furthermore, one must limit the total number of `inodes` in the file cache for each VS. If an `inode` is shared by two or more VSs it should retain the tag of the highest priority VS that is using it. Otherwise, priority inversion would result. Although easy to describe, this feature requires substantial changes to the structure of the file cache. Nevertheless, content servers with very large `inode` working sets would benefit from such insulation. It is possible that this eliminates the small performance degradation of the well-behaved site in Figure 12.

References

[1] AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. Adaptive Algorithms for Managing Distributed Data Processing Workload. *IBM Systems Journal 36*, 2 (1997), 242–283.

[2] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 45–58.

[3] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Retrofitting Quality of Service into a Time-Sharing Operating System. In *USENIX Annual Technical Conference* (June 1999).

[4] BRUNO, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, U.S.A, June 1998).

[5] CAO, P., FELTEN, E. W., AND LI, K. Implementation and Performance of Application-Controlled File Caching. In *First Symposium on Operating System Design and Implementation* (1994), ACM.

[6] CISCO INC. Local Director (White Paper) http://cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/lobal_wp.htm. 2000.

[7] ENSIM CORP. *ServerXchange (White Paper)*. Mountain View, California, 2000. http://www.ensim.com/products/wpaper_fr.html.

[8] F5 CORP. http://www.f5.com/bigip/. 2000.

[9] GOYAL, P., GUO, X., AND VIN, H. M. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Second Symposium on Operating Systems Design and Implementations* (Seattle, Washington, U.S.A, October 1996), ACM, pp. 107–122.

[10] HAND, S. M. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Oreleans, Lousiana, February 1999), USENIX, pp. 73–86.

[11] HYDRAWEB TECHNOLOGIES, INC. Load balancing (White Paper). www.hydraweb.com, 1999.

[12] JEFFAY, K., SMITH, F., MOORTHY, A., AND ANDERSON, J. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).

[13] LAMPSON, B. W., AND REDELL, D. D. Experience With Processes and Monitors in Mesa. *Communications of the ACM 23*, 2 (February 1980), 106–117.

[14] LAURIE, B., AND LAURIE, P. *Apache: The Definitive Guide*, 2nd ed. O'Reilly & Associates, February 1999.

[15] MERCER, C., SAVAGE, S., AND TOKUDA, H. Processor Capacity Resevers: Operating System Support for Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems* (May 1994), IEEE.

[16] OIKAWA, S., AND RAJKUMAR, R. Linux/RK: A Portable Resource Kernel in Linux. In *Work in Progress 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).

[17] POSTEL, J. Internet Protocol Darpa Internet Program Protocol Specification. IETF RFC, September 1981.

[18] SPATSCHECK, O., AND PETERSON, L. L. Defending Against Denial of Service Attacks in Scout. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 59–72.

[19] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECWeb99 (White Paper)*. http://www.spec.org/osg/web99.

[20] SUN MICROSYSTEMS INC. *Solaris Resource Manager 1.0 (White Paper)*. Palo Alto, California. http://www.sun.com/software/white-papers/wp-srm/.

[21] SUN MICROSYSTEMS INC. *Sun Enterprise 10000 Server: Dynamic System Domains*. Palo Alto, California. http://www.sun.com/servers/white-papers/domains.html.

[22] VERMA, D. *Supporting Service Level Agreements on IP Networks*. Macmillan Technical Publishing, 1999.

[23] WOLFF, R. W. *Stochastic Modeling and the Theory of Queues*. Prentice Hall, Englewood Cliffs, NJ, 1989.