Remote Code Browsing, a Network Based Computation Utility

C. Giblin, S. Rooney, and A. Bussani

IBM Zurich Research Laboratory
8803 Rüschlikon, Säumerstrasse 4, Switzerland
cgi@zurich.ibm.com, sro@zurich.ibm.com, bus@zurich.ibm.com

Abstract. *Remote code browsing* is a service by which an end user can select the location of a potentially useful piece of software and have the software fetched and installed on a virtual machine running on a remote server, such that the user may test the suitability of the software without having to install it on their own machine. This capability is made possible by a range of existing technologies combined in a novel way. This paper explains the motivation for code browsing, and describes how our current implementation works.

1 Introduction

This document is being written using a free text editor *emacs*, the postscript is generated with a free text processing software *latex* and both run on a free operating system *Linux*. The value of free software distributed over the Internet is hardly in doubt. However, that value is diminished by the following factors:

- there is always a danger that the downloaded software will disrupt the normal working of the system on which it was installed either because the software is erroneous or the authors were malicious or simply because the person who installed it misunderstood its expected effect. This is particularly true when the installed software changes the functioning of the operating system itself. Moreover, installs are often not easily reversible, i.e. the uninstall does not remove all traces of the install.
- often the only way to determine the precise behavior of a publicly available piece of software is to execute it. But the action of downloading and installing many such packages is labor intensive and frustrating.
- the software may only function on a platform unavailable to the user, however the utility of the software might be the factor which determines whether it is worthwhile changing platform.

Installing the software on a machine other than the user's own removes the possibility of disruption at least to the user. One could imagine an organization might dedicate some machines, running a variety of different operating systems to this task. However, it would not make the act of installing the public software any easier and such machines themselves would be vulnerable to disruption,

M. Feridun et al. (Eds.): DSOM 2002, LNCS 2506, pp. 121–132, 2002.

especially as they would be shared between many members of the same organization.

Ideally, each user would have a dedicated set of machines (each running a different operating system) for installing foreign code. If in addition, some person or people in the organization were responsible for downloading and installing code on the user's behalf, then all the limitations mentioned above would be removed. In general however, this would be prohibitively expensive.

Our code browsing utility goes some way towards automating the task of downloading and installation — thereby removing the need for additional personnel — and using dynamically created virtual machines on which to install the code — thereby removing the need for a large number of physical machines. Application areas are:

- *security analysis*, security specialists can archive and reactivate potentially malicious software in a controlled and safe environment as and when required.
- *software vendors*, vendors can furnish the means by which end-users can execute the trial versions of the vendor's software without having to install it on their machine.
- *software testing*, the tester can dynamically deploy baseline configurations and execute tests on demand. As a result the testing iterations are faster and more dependable.
- *business user*, users can casually browse software packages without local installation.

This paper demonstrates the feasibility of the code browsing service by describing how it works. The code browsing service uses a range of different existing technologies and these are introduced and explained as part of the demonstration.

2 Code Browsing Utility Overview

The code browsing utility is a distributed set of software components which interact in order to achieve the desired code browsing function. These components make use of a range of different technologies including: servlets, tuple spaces, dynamic DNS, DHCP, VNC, virtual x86 architecture, amongst others. In order to combine these different technologies such that they function together coherently, we adopt a very loosely coupled model of interaction.

The code browsing utility divides into two parts: *a download service* with which end-users interact and which is responsible for retrieving the code from the remote site and passing it, along with some meta information, to the virtual machine factory; *a Virtual Machine (VM) factory* which creates the virtual machines and installs the code on them.

The download service copies the file to be code browsed to a file system directory along with a meta information entry. This entry contains the file's name, the operating system on which it should be installed and its size, as well

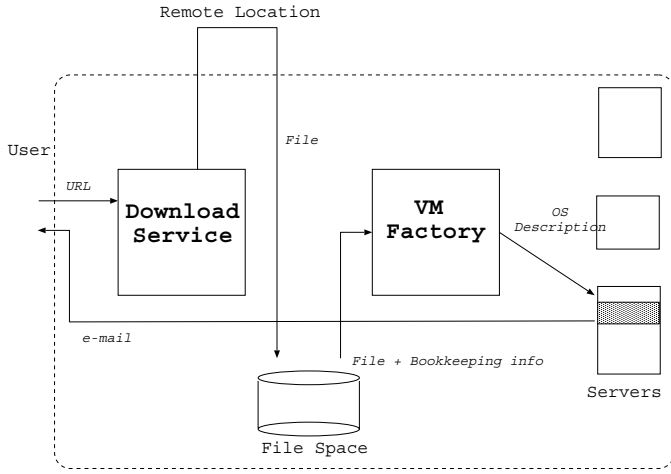

Fig. 1. Overview of the code browsing utility

as information about the virtual machine, e.g. whether it should be persistent between boots, how long it should exist. The VM factory is notified of a new code browsing request and sets about creating the virtual machine with the requested code installed. The VM factory creates a virtual machine of the appropriate type, e.g. Redhat Linux, Windows XP, Windows NT etc. and notifies the user of the system's availability by e-mail. Users then access that virtual machine using remote terminal technology. Figure 1 shows the essential interactions between the main components of the code browsing utility.

3 Virtual Machine Factory

The VM Factory[1] creates virtual machines which are run on a set of physical machines — currently Linux PCs — which constitute the hosting environment. These Linux PCs are connected across an Ethernet LAN to an edge router which links the infrastructure to our intranet.

The VM factory consists of three types of entity: *gatekeepers* which are the entry point of the VM factory and which coordinate the activity of the other components; *server controllers* which create the VMs; *directory services* containing the system's shared information.

For this description, we will assume one gatekeeper and one directory service per code browsing utility, and one server controller per physical machine in the hosting environment.

[1] The VM Factory makes uses of the ICorpMaker control software [1] that automates the addition of new clients to an Application Service Provider's infrastructure. Allowing users to dynamically install arbitrary pieces of software rather than one from a small predefined set is a natural extension of this resource renting model.

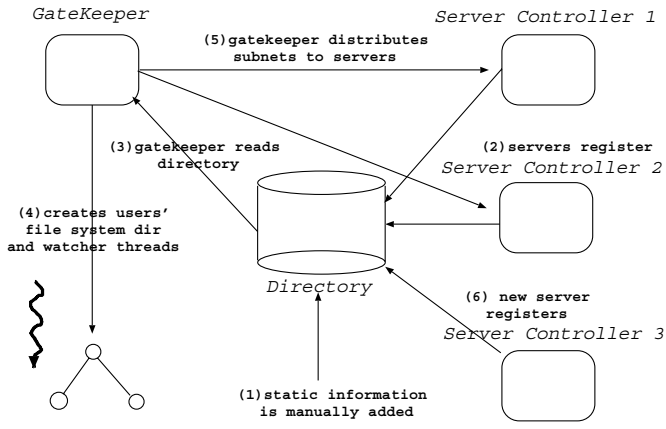

Fig. 2. Starting the VM factory

Dynamic IP Subnet Allocation

Central to the VM factory is the directory service which manages a range of information about the environment. There are two distinct types of information kept in the directory: static information manually entered by a human operator, e.g. registered users, IP addresses to allocate to virtual machines; dynamic information added by the entities themselves.

When the controllers within the hosting environment are started they register themselves with the directory service. When the gatekeeper is started it uses this directory service to look-up the registered code browsing users, creating a file system directory for each user. The gatekeeper then starts a set of watcher threads which monitor the addition of new files.

When new files to browse arrive, corresponding meta information, in the form of entries in a bookkeeping file, is updated. The watcher monitors these entries, interpreting a new entry as a request for a new code browsing session. An alternative would be to have the download manager communicate directly with the gatekeeper, but as we envisage multiple different access methods to our system we chose a decoupled model which reduces run time dependencies. The gatekeeper and the download manager need only to agree on the network location to which downloaded files should be written and the format of the meta data information. Multiple files, for example, multiple interdependent *rpms*, can be installed within the same code browsing session. The order of downloading is the order of installation.

The gatekeeper reads from the directory both the address range which is available for assigning to virtual machines and the servers which are currently registered. It divides this IP range into smaller subnets and allocates one to each of the servers; the server controllers allocate addresses in this range to virtual machines.

Each time that the gatekeeper discovers that it cannot allocate a code browsing session because of a lack of servers, it rereads the directory to see if any new servers have been activated and if so allocates these new servers each a free IP subnet. When the gatekeeper notices that a controller has stopped it removes the IP subnet allocated to that server and will reallocate it to the next controller about which it becomes aware. This means that at different times a given IP address may be allocated to virtual machines running on different physical servers and therefore resolve to different MAC addresses. Some Ethernet switches intercept ARP requests to maintain their own IP/MAC cache, this would cause lost packets if the cache aged slower than the dynamic reallocation of a subnet. In order to flush the cache each time a new virtual machine is activated it sends a single broadcast ping to the entire subnet. Figure 2 shows the various stages in starting the VM factory.

Code Browser Virtual Machine Creation

When a watcher thread detects a new code browsing request, it prompts the gatekeeper to create a code browsing session. The gatekeeper asks each of the registered servers in turn whether they are capable of supporting a new virtual machine of the specified type until either it finds one which is capable or discovers that the request is currently unrealizable. If there is not enough resources within the hosting environment to support the code session, then the watcher thread goes to sleep for a certain amount of time and retries. The amount of time it sleeps is random, but is chosen from a range which increases exponentially after each retry. This Ethernet-like synchronization of competing entities over a shared resource is simple and prevents unfair starvation[2]. If after retrying a certain number of times, the watcher fails to create the session, it sends an e-mail informing the user that they should retry at another moment. Figure 3 shows the steps involved in creating a code browsing session.

If the gatekeeper can no longer communicate with the controller, it assumes the controller has failed and removes the dynamic information about the corresponding server from the directory. Once an available physical server has been identified, the gatekeeper sends it a description of how the virtual machine should be created.

The technology the code browser uses for supporting virtual machines is currently VMWare [2]. VMWare is a commercial software allowing multiple different guest OSs to be run on a given host OS by virtualizing the x86 architecture, similar freeware software is also available [3]. Current work is looking at using dynamically created Linux images on a mainframe [4]. If virtual servers are available for different processors then the user chooses the processor type on which their software should be installed in the download request.

[2] A more complex scheduler could be imagined in which the duration of the code browsing session is a factor in the promptness with which it is scheduled, e.g. shorter sessions first, the rational is that a user who wants a code browsing session for a long period is less likely to need it quickly than one who wants it for a short period, moreover people who are frugal in their requests should be privileged over those that are less so.

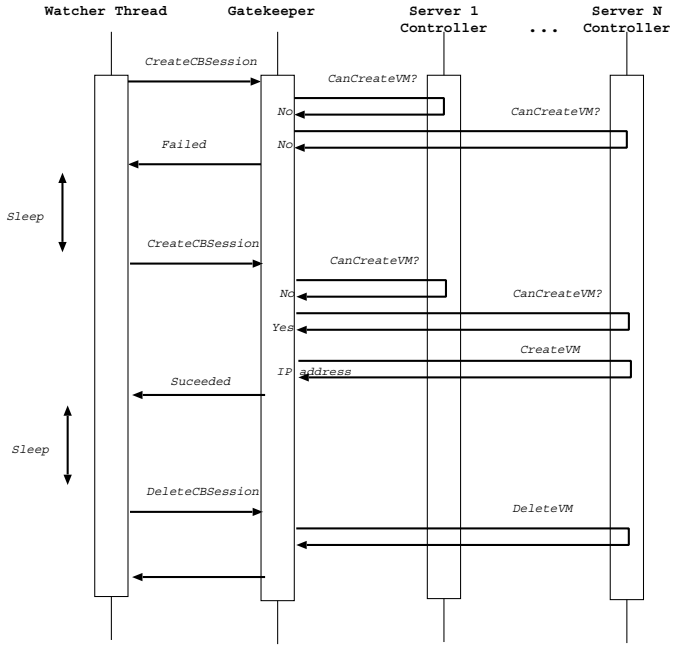

Fig. 3. Creating a code browsing session

The VMWare instances are preconfigured. This involves creating the VMWare virtual disks and installing an OS type on them. In addition, we add a small amount of code in order to allow the guest OS to be controlled within the code browsing utility.

At startup the controller identifies how many configured VMWare instances it has at its disposal from a configuration parameter. The controller will permit a maximum of that number of simultaneously active code browsing sessions. When the gatekeeper assigns the server an IP subnet, the controller uses this subnet and the information about the MAC addresses contained in the VMWare configuration file in order to create a configuration file for the Dynamic Host Control Protocol (DHCP). One DHCP daemon is run per physical server. The controller matches each of the allocated IP addresses to each of the discovered MAC addresses in a fixed order and then restarts the DHCP daemon (*dhcpd*). This happens each time the controller receives an address allocation, i.e. each time either the gatekeeper or the controller is restarted.

During the lifetime of *dhcpd* the same VMWare instance on a given physical server will always be allocated the same IP address; dhcpd behavior remains standard and from the server controller's point of view is predictable.

When the controller is asked by the gatekeeper if it can support a requested code browsing session, it will reply positively if a VMWare instance is free and it supports the appropriate OS type. The gatekeeper then initiates the code

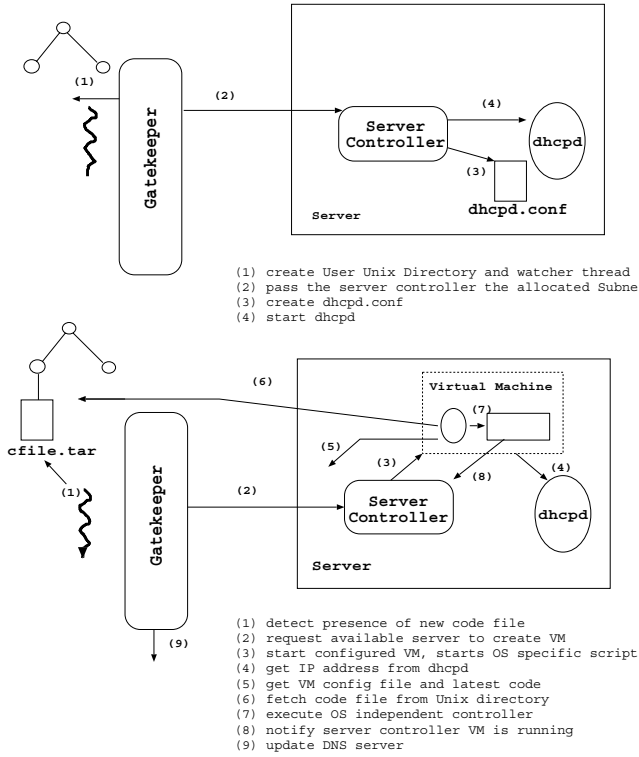

Fig. 4. Key stages in initialization and VM creation

browsing session on the controller. The controller starts by creating a virtual machine configuration file containing information such as the hostname of the virtual machine, the name of the user, the local URL of the code to install.

Once this is done it starts a script which launches one of the VMWare instances. VMWare allows the state of an OS to be checkpointed and resumed. In order to speed up the creation of the virtual machine, we checkpoint the VMWare instance just before the end of its boot process — for example, just before the execution of *rc.local* in Linux — each time VMWare is activated it resumes from this point. Immediately after the checkpoint, a script is executed that is used to configure the virtual machine in the appropriate way. This script copies — using *rcp* — from a well known location the code browsing utility software used to control the virtual machine and the freshly created configuration file. The code browsing initialization is achieved by an OS specific script and an OS independent Java jar file. The OS specific script, after executing the platform specific commands, chains to the platform independent initialization.

The action of the OS specific script depends on the OS: on Linux it creates an account for the user named in the configuration file of the virtual machine

and sets the password. The remote user may then telnet into this account. On Windows it starts the Virtual Network Computing server [5] and sets its password. VNC is a technology which transmits the framestore of an OS across the network much like a video stream. A VNC client is available as a Java applet and hence the remote display can be rendered inside a conventional browser, without change to the client's machine. This allows remote users to "login" to Windows machine, even from a non Windows platform. An alternative would be the use of NetMeeting which we found to perform better than VNC when the user runs Windows locally and accesses a remote virtual Windows system. Figure 4 summarizes some of the key stages in the initialization of the VM factory and the creation of a VM.

Virtual Machine Code Install

With the requested virtual machine now created and initialized, the final step automatically installs the downloaded software into the virtual machine. A script inspects the software to be installed to determine the installation process. For example, the default Linux script simply checks the file for some well known suffixes, e.g. *rpm*, *tar.gz*, *zip* and executes the appropriate commands on them. Likewise the default Windows install script is capable of unzipping and performing a silent installation of the requested package. As a result, when the users logs into the virtual machine, the desired software is already installed. In the worst case the file type is unknown or the installation fails and the file is left for the user to install themselves. The user may also transfer data files usable by software installed in the same code browsing session using the same method.

The user can customize the installation process by supplying their own install script for non generic tasks, e.g. compilation and executing. This script is downloaded as a part of a software package to be browsed and is recognized (by naming conventions) as overriding the default installation. It then starts the virtual machine controller and terminates. After starting VMWare, the controller on the physical machine waits for a notification from the controller on the virtual machine that all is well. The creation action of the server is now complete.

At the moment although our installation process is sufficient for our current prototype, it is somewhat ad hoc. We envisage an architectural entity which manages the installation process taking into account formal requirements specifiable by the user such as software dependencies, in some convenient language, e.g. XML. This entity should also be capable of taking advantage of heuristics in situations in which complex installations are not easily formalized a priori.

Our server farm consists of four PC's running Redhat Linux 7.2 each with 642 Mbytes of main memory, a Pentium III running at 1 Ghz and a SCSI device driver. We found that each PC was able to support 4 VMWare instances each of which emulated a machine with 64 Mbytes of main memory and had a 2 Gbyte virtual disk. Above 4 VMWare instances we found that performance diminished significantly due to excessive swapping. Typically trial applications were: Quake II game server, the video server from Real Networks, WebSphere standalone edition and the Eclipse development environment. All of these ran

acceptably, if not extremely quickly on our infrastructure. A VMWare instance can be configured and started on behalf of the user in under 30 seconds.

User Code Browsing Session

After the successful creation of a VM, the gatekeeper sends a dynamic DNS update to a DNS server that it manages, such that the user's name within the domain of the naming server resolves to the newly allocated IP address. The cache time, Time To Live (TTL), of this new record in the DNS server is set to a very low value — currently one second. In practice we have found that some DNS servers ignore low TTL values and cache the record for a longer period, moreover end-user applications such as web browsers and video players often keep their own cache. To avoid the problem of naming interference, in addition to using the user's name as a DNS entry we also send the IP address of the newly created VM to the user in the form of a URL resolving to the newly created virtual machine.

The watcher thread then looks up the registered user's e-mail address and sends a message informing them that the code browsing session is now available. The thread then goes to sleep for the desired duration, when it wakes up it removes the code browsing session, in a way similar to its creation.

The user receives the e-mail containing a URL to a generated HTML page which contains the information about the code browsing session, including DNS and IP address of the VM, the duration of the session, the OS type of the VM and the code that was installed. It also includes the VNC or telnet prompt which allow the user to enter the created VM.

For virtual Linux system, the user telnets into an account whose name and password are those supplied by the user during registration. The requested code has been placed in this account and unpacked. The user has root privileges on the VM, permitting them, for example, to install kernel patches.

Only one code browsing session may be active per user at a given time, we have found that this encourages users to only ask for a reasonable amount of time for the duration of their code browsing session as they cannot obtain a new one until the old one is complete. During a given session, the user may download new code into their code browsing directory using the downloader; the corresponding session will be activated after the old one terminates.

The user may damage the VM, but this is of no importance, as the VM will cease to exist at the end of the code browsing session. In fact allowing users to first attempt potentially dangerous things in a disposable environment is one of the motivations for the code browsing utility.

Session Termination

After the code browsing session has finished the watcher thread removes the code file from the user's file system directory and places it in a cache directory. This cache is available to users of the download session, who may then choose to install code from the cache rather than from a remote site. This serves to inform the user about software that other users are installing. At the moment the cache is unsophisticated — simply keeping the N most recently downloaded files — but more interesting caches can be imagined for guiding a user's choice, for example

information about the most frequently code browsed software; information about what other users who browsed a specific piece of software also browsed.

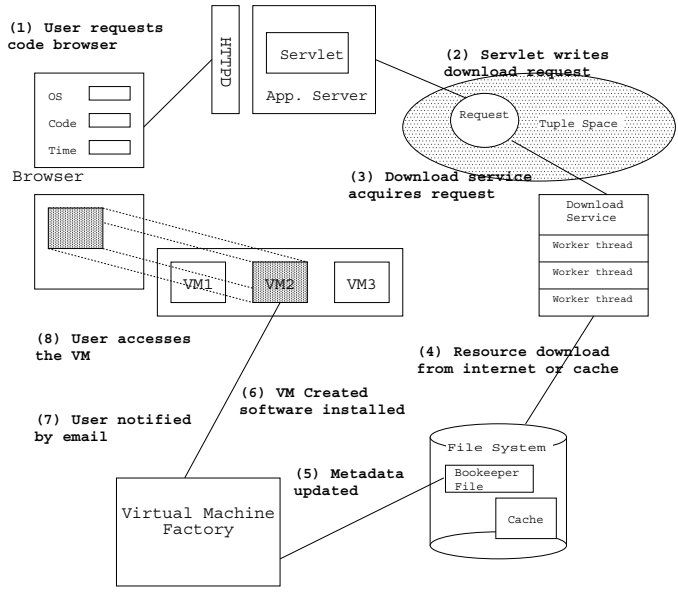

Fig. 5. Download Service

Persistent Virtual Machines

In general, within the code browsing utility VMWare instances are run in non persistent mode, i.e. all changes made during a given run of the OS are lost when it is rebooted, so each time a new session is started the OS is in a clean state. However, the effect of the addition of some new software, for example kernel patches on Linux, are only taken into account when the machine is rebooted. The user may specify that the OS be persistent across reboot, to achieve this we make a copy of a 'clean' VMWare instance making it persistent. At the end of the code browsing session this copy is deleted, i.e. the OS is persistent within a code browsing session but not between code browsing sessions.

The additional time needed both to copy the VMWare instance and to reboot from scratch, rather than simply resuming it from a checkpoint, means the initialization of a persistent code browsing session is significantly longer than that of a transient one.

The ability to save and retrieve snapshots of virtual machines in an archive is a natural extension; future work will combine the code browsing utility with high speed archiving and retrieval technology.

4 Download Service

The Download Service (DS) allows the user to delegate the time consuming task of downloading software to a networked service. It automates the acquisition of software on behalf of a user from the Internet, forwarding it to the VM factory.

The DS is implemented as a process containing a pool of worker threads, each of which connects to a networked persistent tuplespace [6] and waits for a download request. A tuple is a vector of typed fields and a tuplespace is a globally shared memory space in which data is stored and retrieved as a set of tuples using a well defined coordination semantics.

When a download worker picks up a resource request, it decides to retrieve the package from the network or cache. In both cases, the package is placed in the designated user file system directory and the bookkeeping file containing the meta-data is updated. The VM factory recognizes the existence of the new request as explained in Section 3 and the worker thread returns to the tuplespace to await the next request.

The worker threads are stateless and maintain no context common to the requesters of the service or the VM factory. The tuplespace decouples the requestor, the downloader and the VM Factory, making each component independent of the others' availability. Since the tuplespace is persistent and supports transactional semantics, requests survive system restarts and failures.

Scaling to handle increasing workloads is a matter of increasing the number of DS processes. Since the tuplespace is networked, multiple systems can access the tuplespace, forming a pool of distributed and heterogeneous downloader systems.

Although the current system supports downloading over the FTP and HTTP protocols, extending it to support other protocols is straightforward. The worker can be extended with new protocol support through sub-classing and then configured in the DS. Tuplespace queries can be customized for specific request types, allowing system administrators the option of grouping specific protocol support onto dedicated processes or systems. Figure 5 shows how the DS interacts with the rest of the code browsing utility.

5 Discussion/Conclusion

The system described in this paper is fully functional and was made available to fellow researchers at our laboratory for a period. While many people tried it out, ultimately they did not use it for software testing. In our opinion, this was because the investment in learning a new experimental system was not sufficiently compensated by the advantages accrued. The system would be more beneficial if it offered more OS types and allowed snapshots of virtual machines to be save and retrieved.

As well as enhancing the functionality of the code browsing service, we intend to make it available as a Web Service [7]. This would allow it to be a component in other Web Services.

In summary, we view the remote code browsing service as an example of a new type of network based computation utility which is enabled by the quick dynamic creation of virtual partitions of physical resources.

References

1. S. Rooney, "The ICorpMaker, a Dynamic Infrastructure for ASPs," *Proceedings of IEEE Workshop on IP Operations & Management*, Sept 2000.
2. VMWare, "Getting Started Guide, VMWare 2.0 for Linux," *VMWare Technical Support*, January 2000.
3. K. Lawton, "Running multiple operating systems concurrently on an IA32 PC using virtualization techniques," *Part of the plex86 source release, available from www.plex86.org*, November 1999.
4. IBM, "S/390 Virtual Image Facility for Linux, Guide and Reference," *Version 1.0, Release 1.0 SL0500, GC24-5930-04*, June 1999.
5. T. Richardson, Q. S. Fraser, KennethWood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, pp. 33–38, Jan/Feb 1998.
6. P. Wyckoff, S. McLaughry, and T. L. D. Ford, "Tspaces," *IBM Systems Journal, http://www.research.ibm.com/journal/sj/373/wyckoff.html*, August 1998.
7. H. Kreger, "Web Services Conceptual Architecture (WSCA 1.0)," *IBM Software Group white paper*, May 2001.