Virtual Linux servers under z/VM: Security, performance, and administration issues

D. Turk
J. Bausch

In this paper we describe our experience at Colorado State University running hundreds of virtual Linux® servers on an IBM S/390® mainframe with the z/VM® operating system and the way we solved the security, performance, and administration problems that were encountered.

With the large increase in the number of Web-based systems in the last decade, the computational requirements for supporting Web applications grew dramatically. Frequently an organization required dozens, hundreds, or even thousands of servers to support its customer load.[1,2] "Server farms" were created in which machines were used as Web servers, file servers, database servers, and application servers. Although not a new idea—IBM mainframes have had virtual machine capability since at least 1972 when the VM operating system (then known as VM/370) was introduced[3,4]—there has also been a recent focus on running virtual servers on powerful personal or mainframe computers.[5–9] In this paper we describe our experience at Colorado State University (CSU) running hundreds of virtual Linux** servers on an IBM S/390* mainframe[10] with the z/VM*[11] operating system.

The 1980s ushered in the personal computer (PC) era, which provided us with small but powerful machines that could act as servers in the client-server model of computing. The PC came at low cost and allowed end-user control of the system, without interference from a central information technology (IT) department. As client-server computing became popular, many tasks that had been traditionally performed on mainframes were migrated to the smaller and less costly PC servers. In fact, especially with the advent in the 1990s of the freely available Linux operating system and other open-source software such as the Apache Web server, server farms consisting of dozens or hundreds of these commodity PC servers were set up to service heavily used Web sites. The ability to purchase PC servers for only a few hundred dollars drove demand away from the mainframe and to these small, inexpensive, and disposable servers.

Managing large numbers of these servers, however, had its drawbacks. There had to be physical space to store them and physical wiring to interconnect

them. They had to be individually installed, configured, and maintained. It could take significant time and support personnel to keep these server farms running. The low purchase cost of these PC-based servers belied the real total cost of ownership associated with operating these server farms.

In the last few years we have witnessed a return to the mainframe for specialized applications.[12–18] Using the ability of z/VM to host multiple virtual servers running a variety of guest operating systems, mainframes such as IBM's S/390 and zSeries* are now being used to create virtual server farms.[7,16,19,20,] These systems offer high processing speed and high bandwidth for I/O; the physical space they require is comparable to that for hosting small physical server farms; and it is easy to clone an initial server as many times as needed. This cloning can even be automated so that no human intervention is required to carry it out. Thus some of the problems encountered in the physical server farm environment go away. Running virtual Linux guests on a mainframe leads to space savings, easier management, and the benefit of the reliability and security of the mainframe.

One case in which virtual servers are cost-effective is when they are used as Web and database servers.[21–23] In order to support a high-volume Web site, one that is heavily accessed by large numbers of people, it is not uncommon to employ hundreds of virtual servers. In a virtual environment, new servers can be cloned on demand and a load balancer distributes incoming requests among them. Virtual servers use shared resources, such as memory and computational power, and thus are superior to physical server farms in terms of the efficient use of resources.

Because Linux can be used as the guest operating system on a z/VM virtual machine, the large collection of open-source software is readily available and can be run on such a virtual server. Server farms created in this environment can be controlled and managed efficiently on a single physical machine.

The rest of this paper is structured as follows. In the next section we introduce basic z/VM concepts. In the following three sections we describe our experience in running multiple Linux virtual servers on z/VM at CSU and focus on three types of problems we encountered: security issues, performance issues, and administration issues related to managing Linux instances. We conclude with a summary and directions for further work.

z/VM CONCEPTS

z/VM supports the creation of virtual machines by virtualizing the resources of the computing platform: processing, communication, memory, I/O, and networking resources. Although virtual machines share physical resources, they are isolated from each other, and each virtual machine provides the illusion of access to the entire computer. Each virtual machine hosts a guest operating system such as z/OS*, OS/390*, CMS, and Linux. The guests operate completely independently of one another.

Virtualization, such as that provided by z/VM, is often used together with partitioning. On computing platforms such as the S/390 and the z/Series, the hardware supports logical partitioning (a logical partition is referred to as an LPAR) of the machine into several distinct and isolated regions that operate independently of one another. Each unit of physical resource is thus mapped into a specific partition.

Partitioning and virtualization can be viewed as complementary technologies. *Figure 1* illustrates a range of possible configurations for running Linux on an S/390 or zSeries mainframe that involve virtual machines and LPARs.

In Figure 1A Linux is running natively on the mainframe, that is, without logical partitioning. This option is rarely used because it provides for a single operating-system instance on the entire mainframe. In fact, it is no longer possible to configure zSeries mainframes without LPARs—the IBM eServer* zSeries 900 (z900) series was the last family of processors that supported this option. In Figure 1B the machine is configured with three LPARs, and Linux is running natively within each LPAR, as before. In this configuration no virtual machines are created; Linux is running natively and not as a guest operating system on a virtual machine. The maximum number of LPARs depends on the hardware and varies between 15 and 30. Figure 1C depicts a configuration without partitions in which z/VM supports any number of Linux guests (three guests are shown in the figure). The fourth scenario illustrated in Figure 1D combines logical partitioning

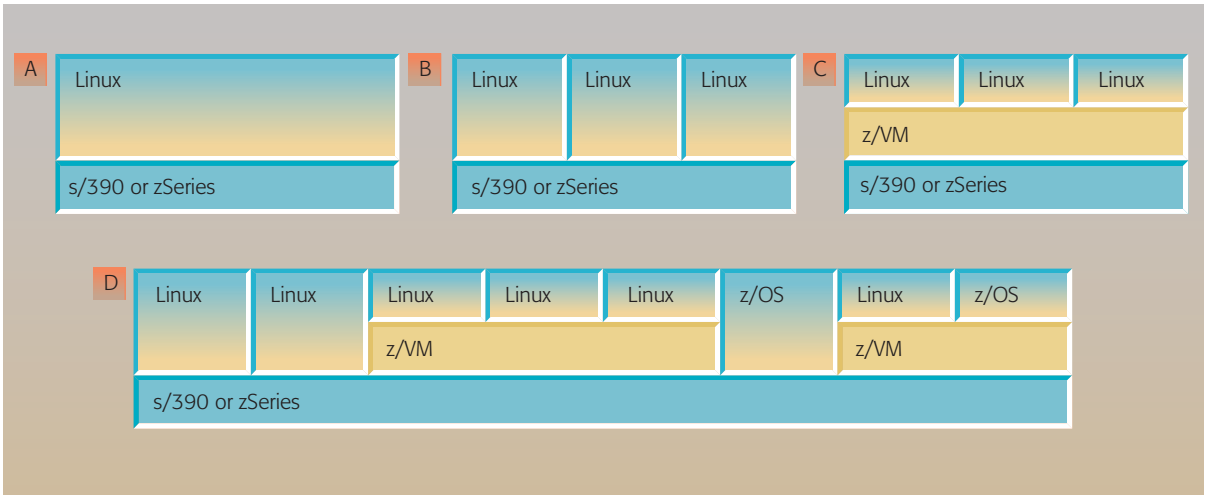

Figure 1
Possible configurations for running Linux on S/390 or zSeries

and virtual machines. The figure shows five LPARs; three of these host operating systems are running natively; whereas, the other two partitions host z/VM instances that support a number of virtual machines. Note that the operating systems running either in an LPAR or on a z/VM-supported virtual machine need not be the same.

The fourth option is similar to the design used for the CSU system. It operates within one of the LPARs and has approximately 500 virtual Linux-server z/VM guests running within this LPAR. Anywhere between 50 and 300 of these are typically active at any given time. At the time the work described here was performed, we were using Version 4.4 of z/VM, running on S/390 hardware, with almost all of the Linux instances running SLES 8 (SuSE Linux Enterprise Server Version 8) and only a few running SLES 7. Although our platform was S/390, most of the results should apply to the zSeries platform as well.

SECURITY ISSUES

In this section we focus on three security-related issues: account passwords, ssh (secure shell) keys, and software updates. Although these are important issues for all servers, for systems with large numbers of cloned virtual servers they become critical.

Account passwords

Access to a Linux instance typically requires an account and a password. In an environment with hundreds of Linux instances, account maintenance can become extremely time-consuming, and thus some form of automated maintenance is needed.

Although a z/VM administrator has the ability to issue commands and push out updates to all Linux instances (z/VM guests), using this approach has its drawbacks. Possibly the biggest problem is the time delay involved. The Linux administrator must determine what updates are needed, request the z/VM administrator to perform the updates, wait for this task to get done, and then inform the Linux-instance users that the updated machines are ready for use. Relying on the z/VM administrator to carry out these updates clearly lengthens the process.

In order to avoid this delay, and to remove much of the dependency on the z/VM administrator, we developed a fully Linux-based solution for automatically performing updates on a large number of instances. This solution relies on ssh/scp (secure shell/secure copy) and a common Linux user account that we created on all Linux instances. The common account has the same login and password for all instances in the system and is given root privileges (that is, it is set up with UID 0). This provides a "back door," separate from the root user, through which all virtual servers can be updated automatically.

Soon after implementing this plan we realized that it had a serious security flaw. When we created a user

account with root privileges and with the same password on all instances, we assumed that the person who had this information would not share it with others. We had overlooked, however, that the owner (administrator) of each Linux instance (not the z/VM administrator who performs the batch updating of all Linux instances) has root access and therefore has access to the /etc/shadow file, which contains all the encrypted account passwords for the machine. The instance owner could take the encrypted copy of the common account password from /etc/shadow and use a brute force[24] password cracker like John the Ripper[25] to decode the password. Although this process might take weeks or months, once the password is deciphered it would compromise the entire environment because all instances have the same common user login and password.

To address this problem we changed the way the common user account could be accessed. The account was stripped of its privileged (UID 0) status and made into a standard user. Then the account was given root access via the `sudo`[26] command. To solve the brute force attack problem, the password was set to a randomly generated 255-character string using blowfish[27] encryption, and then the account was "locked" with the usermod -L user command, thus removing the ability to login in the normal manner using /etc/shadow passwords.[28] To allow login, we decided the best alternative would be to use the public and private keys (described more fully in the next section) built into the OpenSSH tool set. With this method, each instance could retain a copy of the common user account's public key while the private key would be kept on a secured instance with limited access. This secured instance could then be used as the point from which all other instances would be accessed and batch updates performed.

At the time, using this method seemed to be the simplest and most secure way to remove dependency on the z/VM administrator while at the same time enabling easy and secure updating of large numbers of virtual servers in an automated fashion from one of the Linux instances. An alternative approach would be to use public and private keys to allow automatic connection without passwords through the root account. This would eliminate the need to connect through a regular user account and then use sudo to perform commands. If the root user's private key were known only to those who were administering the collection of Linux clones, then this approach would be relatively secure.

ssh keys

In an ssh session a public and private key pair is used to securely exchange a one-time symmetric key, which then is used to encrypt all of that session's communication. One or more of these public and private key pairs are typically stored in the system-wide /etc/ssh directory and $HOME/.ssh user directories on Linux systems. They are generated by tools within the OpenSSH suite and should be unique to an individual host. Of course, the public key portion can and should be made widely available, but the private key portion must be kept absolutely secure and private because it is used to authenticate a person or machine.[29]

We created these keys while building a "golden image" instance from which we cloned all the other Linux instances. Unfortunately, we overlooked the fact that all instances ended up with identical public and private keys; that is, every instance held a copy of the public and private key pair. Needless to say, the security implications could have been serious. By using the private portion of the public-private key pair, a Linux instance administrator or a malicious attacker could capture and decrypt the ssh session symmetric key and thus be able to monitor the traffic to another Linux instance and decrypt all the transmissions for that session. This would be possible because the attacker's instance held a copy of the same private key that was used by each and every host. Any ssh traffic to and from their machine could be captured and read by the attacker, while the users would erroneously believe their communication was secure because they were using ssh. The entire set of virtual servers was at risk.

This security hole also allows for a man-in-the-middle attack, in which the attacker intercepts the traffic to or from another Linux instance, possibly makes changes to it, and passes it on to the intended destination. The two sides are unaware of the third party (man in the middle) who gained access to the information exchanged.

Whereas there is an easy fix for this problem, it is a matter that can easily be overlooked, and, as mentioned above, could have serious consequences. Most sshd init scripts (typically located in /etc/initd on most enterprise-grade installations) can generate

new keys if none are present when the script is run. The solution is therefore to remove the keys from the golden image and allow them to be generated upon the next start of the sshd daemon on each cloned Linux instance. These files are typically found in /etc/ssh and can be easily identified: all keys have "key" in the file name.

Software updates

Updating the software in our environment, especially when security vulnerabilities are identified, is important. Linux for the S/390 is relatively new and only a handful of vendors support it (e.g., Red Hat, Inc.). We noticed that security patches from our vendor were not available early and lagged behind those for other platforms, such as Intel** x86. We could either wait longer for the software updates for our platform or attempt to build the fixes directly from the source code. Waiting for the product release was a big inconvenience at times and not a viable alternative when there were security vulnerabilities involved.

For building our own fixes, it is convenient to have the Linux instances generated as clones. As all instances are replicas of each other, binaries built from the source on one instance will run flawlessly on others; therefore, we keep a control instance (a copy of the golden image) for such purposes and avoid making major changes to it. We use this instance to configure and build software packages from the source and then push out the updates to each instance. (The techniques used to perform the pushing out of updates are discussed in "Making Changes to Large Numbers of Virtual Linux Servers.") Alternatively, we use the compiled source to build an RPM (Red Hat Package Manager) file that installs the binaries. The RPM method is preferable because RPM files keep a record of the software package installation dependencies.

Regardless of the method used, it is vital that attention be paid, not only to the patches for the S/390 platform, but also to the security patches for other architectures. This is done via the mailing lists that all major vendors provide. If S/390 patches are not promptly released, action must be taken to avoid the possibility of a security exposure.

PERFORMANCE ISSUES

Whereas there are many advantages to using virtual servers, there can be performance limitations in such an environment, especially when many instances run with a heavy load or run the same application at the same time. Ideally, when many instances are active, resource usage should be spread out over time rather than concentrated at a particular time. When resource consumption is high there could be "thrashing," a situation in which resources are increasingly expended on swapping jobs and less real work is accomplished. Thus, the ideal workload for virtual server farms involves independent resource requests that peak at different times.

In this section we look at some problems associated with large numbers of concurrently running cron jobs, and CPU-intensive jobs.

cron jobs

Jobs that are executed periodically, cron jobs, are defined in crontab files and scheduled for execution by the crond daemon. Jobs can be run daily, hourly, monthly, or following any other recurring schedule desired. Every distribution of Linux comes with a default set of cron jobs; the specific jobs included depend on the Linux distribution. Examples of cron jobs are: rebuilding the RPM database, rebuilding the man index, updating the slocate index, rotating log files, and cleaning temporary files. Most of these jobs, which are disk I/O-intensive rather than CPU-intensive, can cause significant performance problems in a cloned virtual environment.

Most current Linux distributions try to simplify cron job management by including the script run-crons, typically located in /sbin, /usr/sbin, or /usr/lib depending on the distribution used. The existence of run-crons allows the Linux administrator to place executable scripts inside of a directory instead of having to put each cron job in the /etc/crontab file. Every 15 minutes this script is run in order to check the /etc/cron.hourly, /etc/cron.daily, /etc/cron.weekly, and /etc/cron.monthly directories, and identify the jobs that are to be executed. Running this script on a large number of instances can cause problems. In our environment, with all the system clocks synchronized, every 15 minutes all instances became unresponsive for up to 1.5 minutes because of the high load caused by all of them simultaneously trying to execute this script. Moreover, this is just run-crons checking to see if there are jobs to be run, not actually running any!

The situation worsens when run-crons determines there are jobs to be run. When 500 instances (a

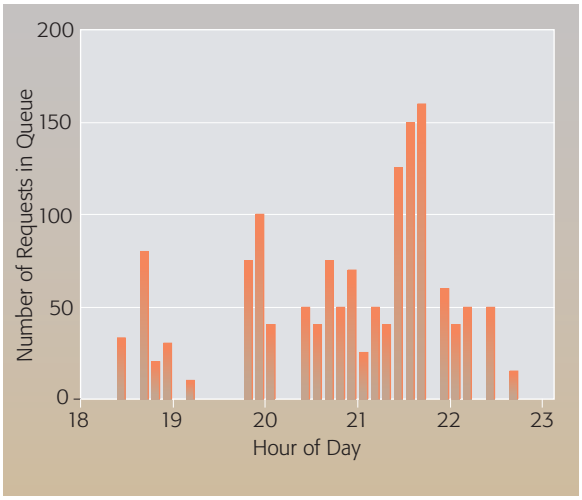

Figure 2
CPU queueing due to execution of cron jobs

typical value at CSU) initiate the same process at the same time this puts the host machine under heavy load. z/VM itself even contributes to the load spike. Because all resource requests cannot be handled simultaneously, z/VM builds an "eligible" queue. When the requests in the queue are processed, each is allocated one (processor) time slice.

Under this load the system becomes unresponsive for long periods of time—in some cases, hours. *Figure 2* shows the system load as the number of requests in the queue due to the execution of cron jobs. (None of this queuing occurs when the cron jobs are not running.) The number of jobs in the queue is plotted against the time of day. Any time when there is significant queuing, the system becomes to some extent unresponsive.

As Figure 2 shows, there are blocks of time where the system would not function well because of this queuing. For instance, we see many spikes above 25, and one spike over 150, between 6:15 p.m. (18:15) and 10:30 p.m. (22:30). Because the queue is relatively large, the response time is very slow, a performance that is not acceptable.

For solving the overload problem caused by cron jobs we use a twofold approach: (1) eliminate unnecessary cron jobs, and (2) avoid the simultaneous execution of jobs by Linux instances whenever possible.

We now describe each one of these approaches in turn.

Eliminate unnecessary cron jobs

Sometimes we are able to eliminate the default cron jobs altogether because, it turns out, most of them are more a convenience then a necessity. These jobs help clean up (e.g., by removing temporary files), perform backup (e.g., backup of configuration files), and help keep the system up to date (e.g., by updating database indexes). In our case, we found that none of the default cron jobs were essential; they were rotating log files, rebuilding the RPM database, and cleaning up temporary files. Although these processes are very helpful, they are not essential.

Eliminating long-running cron jobs is especially useful because their impact is greater. In our case, the job that rebuilt the RPM database was especially problematic because it took the longest to run. When the default cron jobs are removed, the Linux instance owners have to add their own cron jobs when needed. If, however, many system administrators chose to reinstall the default cron jobs, then this solution would fail. The solution would also fail if some of the default cron jobs are deemed necessary and thus cannot be removed from all Linux instances.

By default, the root cron table, /etc/crontab, invokes the run-crons script, which then checks a series of directories for scripts to run as cron jobs. A user on a dedicated server observes the execution of run-crons taking between 0.1 seconds to 2 seconds to run when the files inspected have not been updated since the last execution, and 15 seconds to 2.5 minutes when some of these files have been deleted and need to be updated or when long-running jobs are processed. In contrast, when hundreds of Linux instances cause the simultaneous execution of the script, then the system experiences a significant slowdown.

On our system, the repetitive call to this script every 15 minutes was a problem. It performed extra processing and disk I/O that caused queuing and a serious increase in response times for all our users. With 500 active instances, it resulted in an average of 1.5 minutes of time where the system was totally unresponsive and an additional 2.5 minutes when the system was very slow. Having such a slowdown

every 15 minutes is not acceptable. It is preferable to change the /etc/crontab file to directly execute the script at the time the system administrator deemed appropriate. Using this "direct" crontab method removes some of the overhead associated with the run-crons script. It is worth the extra effort to run cron jobs directly when contrasted with the minimal convenience run-crons gives and the resulting slowdown that occurs.

Avoid the simultaneous execution of jobs by Linux instances

After removing all nonessential cron jobs, we vary the times at which the remaining jobs would execute so as to avoid simultaneous execution of these jobs. The steps involved are: (1) determine the maximum number of instances that can simultaneously execute the cron jobs without causing excessive queuing, and (2) create an execution plan to allow no more than the maximum number of instances to be running these cron jobs at the same time.

After eliminating the unnecessary cron jobs and running the remaining ones directly rather than through run-crons, we tested and timed the execution of these scripts using the time command. The results were placed in a file in /tmp. We first had to find the point where so many virtual servers were executing the cron jobs at the same time that it caused z/VM queuing to occur. We achieved this by gradually increasing the number of active instances and by determining the point when queuing becomes excessive.

We configured 110 instances on which we would run the scripts and synchronized the time on all these machines. The rest of the 500 instances were left idle. Thirty seconds after the designated execution time the system became unresponsive, and the system was left virtually unusable for a period of 1 hour and 37 minutes. ***Figure 3*** shows the results of our experiments as a function of the length of the unresponsiveness period as a function of the number of active instances. When the system became responsive again, minor forensics were carried out by examining the generated time files. The average time that it took to execute the cron jobs on a single instance was just under 3 minutes. With all 110 instances attempting to execute the cron jobs at the same time, all instances slowed down considerably. Twenty-six showed somewhat reasonable times—in the range of 6–10 minutes each.

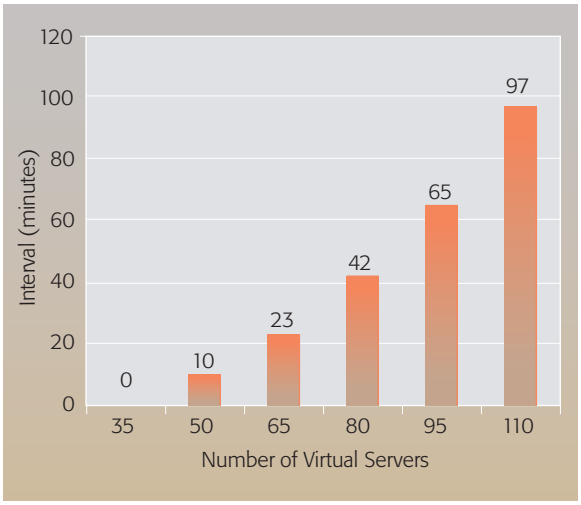

Figure 3
Unresponsiveness interval due to the simultaneous execution of cron jobs

An additional block of 17 took an average of 25 minutes for each instance. This trend continued until the longest execution time was reached, 105 minutes, and the jobs were done.

The same process was repeated five times, each time decreasing the number of instances tested by 15. At 35 instances we found there was no unresponsiveness—the cron jobs finished running in about three minutes, and no queuing was encountered. At this point we started adding one instance at a time and repeating the test, until we reach 42 instances where we found that queuing resumed.

After we determined the load the system could handle, we needed to customize the cron jobs to stay within this load level. The crontab file which we had modified earlier made this easy to do.

From our earlier testing we had found that there was no queuing when we had 35 virtual servers running cron jobs simultaneously. To be conservative, we divided this number in half and then backed off a little, taking 15 as the number of instances that can run cron jobs simultaneously. Because this number was far from the queuing threshold, we were confident that it would not hinder the mainframe's normal performance and would allow normal use for all users. It was also a large enough block so that all the instances would complete their cron jobs within a reasonable amount of time, 6 minutes in

this case. Allowing for some safety margin, groups of 15 instances were programmed to execute their jobs at 12-minute intervals. Using this schedule all 500 virtual servers could complete the cron jobs within a seven-hour time period during the night, with the blocks of instances staggered and thus not causing the queuing and slowdown to occur. The contents of /etc/crontab are illustrated below, with each different crontab entry initiating the running of the cron jobs on a different block of instances at a different time of the day.

```
Crontab 1—for the 1st block of 15 instances:
#will execute at 11:12 pm (23:12) every day.
12 23 * * * /path/to/your/script
Crontab 2—for the 2nd block of 15 instances:
#will execute at 11:24 pm (23:24) every day.
24 23 * * * /path/to/your/script
Crontab 3—for the 3rd block of 15 instances:
#will execute at 11:36 pm (23:36) every day
36 23 * * * /path/to/your/script
...
Crontab 9—for the 9th block of 15 instances:
#will execute at 1:48 am (01:48) every day
48 1 * * * /path/to/your/script
...
```

A different version of the crontab file was pushed to each block of 15 instances. The system queuing problem was alleviated (no process slices showed up in the "available" queues), and we still had the convenience of using cron jobs.

More recently we have developed a way to automatically generate these crontab files with the time offsets generated directly from the IP (Internet Protocol) addresses of the virtual Linux instances and the total number of instances.[30]

Control CPU-intensive processes

Some applications have high CPU requirements. In other cases "looping" can occur in which the job runs indefinitely unless interrupted by the system. When this happens, all other processes slow down almost to a stop.

To prevent these situations from impacting the performance of other instances, we created a simple but effective script that identifies the processes whose CPU use is excessive. When FCONX (an IBM monitoring utility) detects high CPU usage by an instance, it calls a script that kills any process using more than 75 percent of the CPU. The script, written in perl, is available on our Web site.[31]

ADMINISTRATION ISSUES

The automated updating of large numbers of virtual Linux servers is facilitated by tools such as ssh and scp (part of the OpenSSH suite). The ssh command, which uses public key authentication, has two uses. It can be used simply to login and gain access to an interactive command shell on a remote machine and also to run commands on a remote machine. The ssh and scp commands provide rexec functionality in a secure manner. The scp command, which also uses public key authentication, is used to copy files securely from one location to another.

The sshTool script, which we wrote to help manage our virtual Linux instances, is a tool based ssh and scp that performs remote Linux management tasks.[31] This script reads IP addresses (or host names) and a command string from a file and accepts as arguments local and remote paths of a file. Using these three inputs the script can then run the command on or push the file to all the specified instances. A Linux administrator with moderate shell-scripting skills can quickly configure sshTool to push out updates. However, there are a couple of items that must be configured on the servers before this will work.

As mentioned in the section on account passwords, a common Linux user account with the ability to gain root access is needed on all instances. The sudo utility is a good method for providing root access because it can be used to control what the user can and cannot run and because it logs the commands used in the process. This user must be authenticated through ssh trusted keys. As we saw above, trusted keys have great benefits for secure access and also remove the need to use a password at login. Authentication without password is vital to automation because automatic execution of commands is not practical when a password needs to be entered manually for each of several hundred virtual Linux instances.

The final setup in configuring such an automated approach for updates is to address strict host key checking. The strict host key checking configuration indicates how much ssh needs to know ahead of time about the machines it is attempting to connect to, and to what extent the administrator is willing to

allow ssh to update on-the-fly at connection time. Strict host key checking can be set to YES, NO, or ASK, and can be configured permanently with the StrictHostKeyChecking parameter in the ssh_config file or temporarily, just for the current connection, by using the -o flag on the ssh command line. The ssh_config file is typically found in /etc/ssh but may be found other places (such as /usr/local/ssh), depending on how ssh was installed.

A StrictHostKeyChecking value of YES indicates that keys for all hosts to which you wish to connect must exist ahead of time in the system-wide ssh_known_hosts file or the user-based version of this file, known_hosts, that may also exist in $HOME/.ssh for each user. If the key of the host to which ssh is attempting to connect is not in either of these known_hosts files and StrictHostKeyChecking is set to YES, then the connection will be denied. If StrictHostKeyChecking is set to NO, then the key of the host to which ssh is attempting to connect will automatically be added to the known_hosts file if it does not already exist or will automatically be updated if it is different from one in the known_hosts file, and the connection will be made. If StrictHostKeyChecking is set to ask, then ssh will prompt the user as to whether this key should be added to the known_hosts file, and, if so, will add it and proceed with the connection. A good summary of how to set up host keys has been written by Hatch.[32]

The safest way to do this would be to capture all known and verified host keys ahead of time and to store them all in the system-wide ssh_known_hosts file on the Linux instance from which the updates are to be pushed. Having done this, one could then set StrictHostKeyChecking permanently to YES in the ssh_config file and make any connections safely and securely. Less safe, though workable, would be to temporarily set StrictHostKeyChecking to NO and assume that the host's keys have not changed. This, though, opens up the possibility that a key may have been maliciously changed on a target host, or that a man-in-the-middle attack has occurred. In either case, with StrictHostKeyChecking set to NO, the connection is allowed and may be vulnerable to a hacker's attack. Setting StrictHostKeyChecking to NO allowed us to easily automate updates across many Linux instances; however, in order to do this, we had to assume that we knew the host keys had not been changed, either in a valid way or maliciously. Storing of these host keys and the possible reduction in strict host key checking should only be done on one instance—the instance from which the sshTool script is to be used. Having a single instance from which this script can be run is desirable in order to minimize the security exposure.

Once these tasks are completed, it is easy for a Linux administrator to automatically push updates to any or all of the instances. And, at the same time, the Linux administrator is now less dependent on the z/VM administrator for maintaining the virtual Linux servers.

CONCLUSIONS

Running Linux on an IBM mainframe is still a fairly new development. We have addressed in this paper some of the more common challenges for this environment: security, performance, and administration.

We have shown that attention must be paid to the problem of contention associated with the sharing of resources. A well-designed plan must be put in place to ensure that simultaneous requests by a large number of instances for computing resources are avoided. Otherwise, performance may quickly degrade.

Security is another issue that makes the virtual server environment different from physical server farms. Because of the existence of multiple, almost identical, cloned servers, new ways of thinking about protection are required. Compromised security on one server might compromise the security of all other servers. Some issues that seem insignificant at first may become large problems in the future. Because of this, it is important to implement an automated system for maintaining the multiple instances. It is not a complex task, and it brings great benefits when implemented. The `ssh` and `scp` commands, and the use of public and private keys, were shown to be useful tools in this implementation.

When the servers are clones of each other and execute independently, then we have efficient use of resources. Managing the virtual servers is easier, as for example, when additional servers are to be allocated in an on demand environment. These features have the potential for reducing the overall cost of running and maintaining a server farm.

ACKNOWLEDGMENTS

For the first two and a half years of this project, Gerri Peper of IBM inspired, led, helped trouble-shoot, and was the major driving force for the Linux Hub at CSU. Scott Rohling of IBM provided support and maintenance for z/VM and Linux throughout this project, and as this paper was being written, provided very helpful clarifying information regarding z/VM, LPARs, queuing, demand paging, and context shifting in this environment. Bob Barkie of IBM collected data on and helped clarify questions regarding the queuing processes we observed as we solved some of the problems described in this paper. The former and current CSU students who spent countless hours doing the actual work and solving the problems we encountered include Marc Duggan and Yuntao Liu (now with IBM), Yongjian Xu (who was the first Linux support person for our Linux Hub projects, now with IBM), Xiang Zhou, Adrianne Reynolds, Greg Woberman, Damian Jakubczyk, Ram Ramamurthy, Paul Rennix, and Corey Axtell (who is the current Linux support person for our Linux Hub projects). The project was made possible by the support, encouragement, and many hours of legwork provided by the department chairman, Dr. John Plotnicki. This paper has benefited from the valuable comments provided by the anonymous reviewers.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Intel Corporation, or Linus Torvalds.

CITED REFERENCES AND NOTES

1. In April 2004 the number of dual-CPU machines that power the Google service was estimated at over 60,000: http://www.tnl.net/blog/entry/How_many_Google_machines.
2. G. Anthes, "Cigna Goes Virtual," *Computerworld* **38**, No. 32, 23–24 (August 9, 2004).
3. *VM History and Heritage*, IBM Corporation, http://www.vm.ibm.com/history/, Visited Dec 14, 2004.
4. *The IBM Mainframe (including IBM s/360 history)*, IBM Corporation, http://www-1.ibm.com/servers/eserver/zseries/timeline/, Visited Dec 14, 2004.
5. R. Norton, "Using Virtual Linux Servers," *IEEE Computer* **35**, 106–107 (November 2002).
6. M. Rosenblum, "The Reincarnation of Virtual Machines," *Queue* **2**, No. 5, 34–40 (July/August 2004).
7. *Linux at IBM: Server Consolidation*, IBM Corporation, http://www-1.ibm.com/linux/lnxscon.shtml, Visited Dec 13, 2004.
8. G. Geiselhart, et al, *Linux on IBM eServer zSeries and S/390: Large Scale Linux Deployment*, IBM Corporation, http://www.redbooks.ibm.com/redbooks/pdfs/sg246824.pdf, (October 2002), Visited Dec 14, 2004.
9. J. Evers, "Beta Testing of MS Virtual Server 2004 Begins," *Computerworld* (Feb 19, 2004), http://www.computerworld.com/softwaretopics/os/story/0,10801,90324,00.html, Visited Dec 14, 2004.
10. IBM has donated the use of the S/390 since the summer of 2001. IBM has housed and maintained the hardware in Boulder, Colorado, and has provided z/VM support; the authors and other students have provided the Linux support. Access to the system from CSU is via a virtual private network (VPN) over the Internet.
11. *z/VM*, IBM Corporation, http://www.vm.ibm.com/, Visited Dec 14, 2004.
12. R. Scheier, "Moving into Mainframe Linux," *Computerworld* (March 2003), http://www.computerworld.com/softwaretopics/os/story/0,10801,78050,00.html, Visited Dec 14, 2004.
13. A. Bednarz, "IBM Bolsters Its Mainframe Platform," *Network World* **21**, No. 42, 17–18 (October 18, 2004).
14. R. Mitchell, "Master of the Mainframe," *Computerworld* **38**, No. 43, 43 (October 25, 2004).
15. J. Fontana, "Microsoft Mixes Mainframe Apps with .Net," *Network World* **21**, No. 43, 21 (October 25, 2004).
16. C. Saran, "IBM Gives Mainframes New Lease of Life with Reference Architectures," *Computer Weekly*, p. 5 (October 12, 2004).
17. J. Burt, "Big Iron is Back," *eWeek* **21**, No. 41, 36 (October 11, 2004).
18. "Revenge of the Dinosaurs," *Economist* **360**, No. 8231, 52–53 (July 21, 2001).
19. *IBM Virtual Hosting*, IBM Corporation, http://www-1.ibm.com/services/us/index.wss/so/ebhs/a1000349, Visited Dec 13, 2004.
20. E. Scannell, "IBM Achieves Virtualization," *InfoWorld* **26**, No. 18, 22 (May 3, 2004).
21. J. Cummings, "7 Ways to Slash Costs," *Network World* **21**, No. 34, S24–S25 (August 23, 2004).
22. K. Fogarty, "When It Makes Cents to Buy a Mainframe," *Baseline*, No. 30, 76 (May 2004).
23. E. Scannell, "IBM Achieves Virtualization," *InfoWorld* **26**, No. 18, 22 (May 3, 2004).
24. Brute force password crackers typically use large files of common words that people might use for passwords. By systematically encrypting each word in the dictionary with the standard encryption algorithm and comparing it with each entry in the password file, systems can be broken into.
25. *John the Ripper password cracker*, Openwall Project, http://www.openwall.com/john/, Visited Dec 13, 2004.
26. A Linux system administrator can use this command to give controlled administrative access to non-root users.
27. B. Schneier, "The Blowfish Encryption Algorithm," http://www.schneier.com/blowfish.html, Visited Dec 13, 2004.
28. The execution of `usermod -L` puts a "!" character at the beginning of the password field in the /etc/shadow file, thus changing the encrypted password and effectively locking the account from user access.
29. *Crypto FAQ*, RSA Laboratories, http://www.rsasecurity.com/rsalabs/node.asp?id=2152. A good tutorial on cryptography and how public and private keys are used for authentication. Visited Oct 21, 2004.
30. Contact the first author for further information about this approach.

31. D. Turk and J. Bausch, personal Web page, Colorado State University, http://www.biz.colostate.edu/faculty/dant/pages/papers/IBMSysJ/2005-LinuxOnTheMainframe/index.htm, Visited Dec 14, 2004.

32. B. Hatch, "SSH Host Key Protection," SecurityFocus, http://www.securityfocus.com/infocus/1806, Visited Oct 21, 2004.

Accepted for publication November 12, 2004.
Published online April 26, 2005.

Daniel Turk
Computer Information Systems Department, 154 Rockwell Hall, Colorado State University, Fort Collins, Colorado 80523–1277 (dan.turk@colostate.edu). Dr. Turk received an M.S. degree in computer science from Andrews University in 1988 and a Ph.D. degree in business administration (computer information systems) from Georgia State University in 1999. He is currently an Assistant Professor in the Computer Information Systems department at Colorado State University in Fort Collins, Colorado. His research interests are in the areas of computer networking, object-oriented systems, software engineering, business- and system-level modeling, software-development-process modeling, the value of modeling, and process improvement. He is a member of the IEEE and the ACM and has published papers in *IEEE Transactions on Software Engineering*, *L'Objet*, *The Journal of Systems and Software*, *Information Technology & Management*, and the *International Journal of Human Computer Studies*.

Jonathan Bausch
Threat & Vulnerability Management, PricewaterhouseCoopers, LLP, 800 Market Street, St. Louis, Missouri 63101 (jbausch@gmail.com). Mr. Bausch received a B.S. degree in business administration with a concentration in computer information systems from Colorado State University in 2004, where he worked extensively on the Linux Hub project during his senior year. His main interests are in computer networking, Linux and open-source computing, and network security. He is currently a threat-and-vulnerability specialist at PricewaterhouseCoopers. ■