Linux clustering with MOSIX

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

Clustering is a term that is rapidly gaining popularity in the Linux world. But what exactly is clustering, how does one go about cluster-enabling a Linux system, and how can one benefit from setting up a cluster? In this tutorial, Daniel Robbins helps to answer these questions by stepping you through the process of setting up your own MOSIX cluster. MOSIX is a special transparent form of clustering that is very easy to set up and can produce positive results with only a minimal investment of time and energy.

About the author

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org*.

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of *Gentoo Technologies, Inc.*, the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed*, and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. Introducing MOSIX

Introduction

Welcome to the IBM developerWorks MOSIX clustering tutorial! In this tutorial, I will give you a very gentle introduction to clustering technologies available for Linux, and even step you through the process of setting up your own Linux cluster using MOSIX. Clustering technologies allow two or more Linux systems to combine their computing resources so that they can work cooperatively rather than in isolation.

If you're interested in learning more about clustering, then this tutorial is for you. You are likely to find it of great benefit even if you just want to familiarize yourself with Linux clustering technology, but not actually set up a cluster yourself. But if you *would* like to start playing with clustering technology, this tutorial will provide you with a really easy and painless way to do so. So sit back, click away, and enjoy!

Prerequisites

In this tutorial, we're going to set up our own MOSIX cluster. This cluster will consist of two or more Linux systems which we will call "nodes". To set up your own test cluster, you will need at least two Linux systems ready to run or already running kernel 2.4. These systems must be connected to a local area network.

Recommendations

For maximum cluster performance, you may want to consider building a cluster using the following components. These items are not absolutely required; some are completely optional (only for performance freaks), and others are recommended. I indicate which are which below:

At least 100Mbit (fast) ethernet is recommended. Standard (10Mbit) ethernet won't give you very good cluster performance, but should be fine if you just want to play around with MOSIX.

Recommendations, part 2

A good amount of swap space is recommended. This will allow nodes to be removed from your cluster without causing the existing nodes from running out of virtual memory. Again, this is recommended and will only make a difference in extreme situations where you are pushing your cluster very hard.

Gigabit ethernet is optional but beneficial. Gigabit ethernet cards are also dropping in price; reliable ones can be found for $130 to $180 USD. However, don't feel that you absolutely *need* Gigabit ethernet; MOSIX can do just fine with fast ethernet.

Recommendations, part 3

Hooking your machines' ethernet cards up to a dedicated high-speed switch is beneficial. By doing so, your systems will be able to communicate over ethernet in "full duplex" mode, effectively doubling bandwidth.

If you have a limited number of machines, you may want to consider using a specially-wired ethernet cable to directly connect the systems to one another. By doing so, you can benefit from "switch-like" full-duplex performance at a potentially lower price. This trick is very helpful when used for 2 or 3-node clusters, since these configurations only require one or two NICs per machine respectively.

Don't get scared

Again, these suggestions are completely optional, and it is entirely possible to set up a cluster using two Pentium-class machines over a standard ethernet network. Generally, the faster your network, the better MOSIX will be able to migrate processes between nodes in your cluster, and the more fun and exciting MOSIX will be when you play with it. :)

Section 3. The technology behind MOSIX

What is clustering?

Now that I've explained the prerequisites for setting up a MOSIX cluster, let's get a better understanding of what "clustering" is all about. In general, when people speak of "clustering" technologies for Linux, they are referring to technologies that allow multiple computers to work together to solve common computing problems. The computing problems in question can be anything from complex CPU-intensive scientific computations to a simple horde of miscellaneous processes with no overall similarity.

What are Beowulf clusters?

While clustering is a generic term, there are several different clustering technologies available under Linux, and they operate in very different ways.

Probably the best-known type of Linux-based cluster is the *Beowulf* cluster. A Beowulf cluster consists of multiple machines on a local area network that pool resources to solve computing tasks. In order for this cooperation to take place, special cluster-enabled applications must be written using clustering libraries. The most popular clustering libraries are PVM and MPI, and they are both very mature and work very well. By using PVM or MPI, programmers can create cluster-enabled applications that are able to take advantage of an entire cluster's computing resources, rather than being bound to a single machine.

Drawbacks of Beowulf

While Beowulf clusters are extremely powerful, they aren't for everyone. The primary drawback these types of clusters is that they require specially-designed software (written to hook into PVM or MPI) in order to take advantage of the cluster. This is generally not a problem for those in the scientific and research communities who have the resources to write their own PVM or MPI code. But those of us who simply want to set up a cluster and gain some kind of immediate benefit have a very real problem -- even if we *do* set up a Beowulf cluster, we won't have any software that can take advantage of it!

The MOSIX solution

So far, clustering may sound like a big disappointment. However, don't get discouraged. Thankfully, there's another kind of clustering technology that's easy to set up and can provide an immediate benefit, and this clustering technology is called -- you guessed it -- MOSIX.

MOSIX works in a fundamentally different way that PVM or MPI, extending the kernel so that *any* standard Linux process can take advantage of a cluster's resources. By using special adaptive load-balancing techniques, processes running on one node in the cluster can be transparently "migrated" to another node where they may execute faster. Because MOSIX is transparent, the process that's migrated doesn't even "know" (or *need* to know) that it is running on a remote system. As far as that remote process and other processes running on the original node (called the "home node") are concerned, the process is running locally.

The MOSIX solution, continued

Because MOSIX is completely transparent, no special programming is required to take advantage of MOSIX's load-balancing technology. In fact, a default MOSIX installation will automatically migrate processes to the "best" node without any user intervention, making MOSIX an ideal turn-key clustering solution that can be of great benefit to very many people.

Mosix vs. SMP

The really great thing about MOSIX is that it can turn a bunch of Linux machines into something like a large virtual SMP system. However, there are a few things that you should keep in mind. First, on a "real" SMP system, two or more CPUs can exchange data very quickly; but with MOSIX, the speed by which nodes can communicate is determined by the speed and type of your local area network.

The "plus" side of MOSIX is that you can create clusters consisting of tens or even hundreds of nodes, while SMP systems generally only go up to two or perhaps four processors. And larger SMP systems can be prohibitively expensive for many people. With MOSIX, you can build a "virtual" SMP system using inexpensive commodity PC hardware.

Mosix limitations -- processes

In addition, MOSIX, like an SMP system, cannot cause a single process to run on multiple CPUs at the same time. So, MOSIX won't be able to speed up a single process such as Netscape, except to migrate it to a node where it can execute most efficiently. In addition, MOSIX can't currently allow multiple threads to execute on separate systems.

The strength of MOSIX

Despite these limitations, MOSIX can have a very immediate and dramatic performance impact. For example, if an application is designed to fork() many child processes which perform work, then MOSIX will be able to migrate each one these processes as needed. So, if you wanted to compress 13 digital audio tracks simultaneously (as separate processes), then MOSIX will allow you to immediately benefit from the power of your cluster. If, However, you were to run the 13 encoding processes linearly (one after another), then you wouldn't see any speedup at all. In fact, we'll take a look at a CD-encoding MOSIX test a bit later in this tutorial. But first, let's get MOSIX up and running.

Section 4. Installing MOSIX

Installation overview

MOSIX installation is relatively painless. To get MOSIX up and running, we must first install a MOSIX-enabled kernel on all the Linux systems that will be joined together into a single cluster. Then, we need to install the MOSIX user tools. Finally, we must make the necessary changes to each nodes' system configuration files in /etc, and then reboot. When the systems come back up, MOSIX process migration will be active and your cluster will be ready for use.

Downloading MOSIX sources

To start the MOSIX installation, first head over to *mosix.org*, and click on the *download and install the latest release of MOSIX* link. On this page, you'll be able to see the various versions of MOSIX available. We'll be using MOSIX for the 2.4 kernel, although there is a version of MOSIX available for the 2.2 kernel as well.

Under "The latest distribution" section, you should see a link to download MOSIX 1.x for kernel 2.4.y; x and y will vary as new kernel and MOSIX releases are made available. Go ahead and download the MOSIX tarball; at the time this article was written, the most recent MOSIX source tarball was MOSIX-1.5.2.tar.gz.

MOSIX and the kernel

Now, make a note of the Linux kernel version with which this particular version of MOSIX was designed to operate. You *always* want to pair MOSIX with the correct Linus kernel to ensure stable operation. In addition, it's highly recommended that you *do not* apply any additional non-trivial patches to your Linux kernel besides the MOSIX patches themselves. Once you've downloaded the correct MOSIX tarball, go ahead and download the appropriate 2.4 kernel from *http://www.kernel.org/pub/linux/kernel/v2.4/*. The version of MOSIX that I'm using over here (1.5.2) was designed to work with Linux 2.4.13, so I went ahead and downloaded linux-2.4.13.tar.bz2.

Patching the kernel

After you've downloaded the correct kernel for your version of MOSIX, it's time to extract your kernel sources and apply the MOSIX patch. Here's how.

```
# cd /root # tar xzvf /path/to/MOSIX-1.5.2.tar.gz # cd /usr/src # mv linux linux.old (if linux is a directory) # rm linux (if linux is a symlink) # cat /path/to/linux-2.4.13.tar.bz2 | bzip2 -dc | tar xvf -
```

Now, we apply the MOSIX patch:

```
# cd /usr/src/linux # patch -p0 < /root/MOSIX-1.5.2/patches.2.4.13
```

Voila! Now that the patch is applied is applied to the kernel sources, we're ready to configure, compile and install a new MOSIX-enabled Linux kernel.

Kernel configuration overview

Now, it's time to configure the kernel:

```
# cd /usr/src/linux (if you aren't there already) # make menuconfig
```

You'll be greeted with the blue Linux kernel configuration screen. Go ahead and configure your kernel with the options it needs to run on your underlying hardware. When you're done, head over to the new "MOSIX" configuration section, which should be the first configuration category listed. In the following panels, I'll explain all the important MOSIX options:

```
[*] MOSIX process migration support
```

This option enables MOSIX proper. It's required.

Complex network option

```
[ ] Support clusters with a complex network topology
```

Enable this option if the nodes on your cluster are not on a simple LAN, or if the network cards you are using in your nodes vary widely in their performance characteristics. If you are using similar or identically-performing NICs throughout your cluster, and all machines are an equal "distance" away (from a network perspective), then you should leave this option disabled for increased performance.

Kernel diagnostics option

```
[*] MOSIX Kernel Diagnostics
```

The MOSIX kernel diagnostics option tells the MOSIX code to perform some additional internal checks. You can enable it for additional safety or disable it for a performance increase.

Security and MFS options

```
[*] Stricter security on MOSIX ports
```

This option will guard MOSIX's TCP/UDP ports against being abused by people on hosts outside the cluster. This is highly recommended if your systems are addressable by systems that are not part of your cluster.

```
[*] MOSIX File-System
```

This enables the MOSIX filesystem, a handy cluster filesystem that can be used to share and copy files throughout your cluster. MFS is very highly recommended and is a very handy addition to MOSIX.

DFSA option

```
[*] Direct File-System Access
```

Enabling this option can allow increased performance for processes migrated away from their "home" node. It's a nice extension, but is still considered experimental. You can choose

to enable it for improved performance in certain situations, or leave it disabled.

Finishing up config

Now, go ahead and save your kernel configuration options. An important note: *make sure that your MOSIX configuration options are identical throughout your entire cluster. Other kernel options may vary due to hardware differences, but MOSIX will expect all nodes to have identical MOSIX functionality. In addition, all nodes of your cluster must use the same version of MOSIX and the Linux kernel. Besides this requirement, feel free to mix different Linux distributions.*

Kernel installation tricks

Now, compile and install the kernel you just configured. Repeat this process on all remaining nodes in your cluster. You may want to copy your /usr/src/linux directory over to the other systems to speed things up. Otherwise, copy the /usr/src/linux/.config file from your current node to any new nodes before typing "make menuconfig". That way, you'll start with your previous node's kernel configuration, ensuring that all your MOSIX kernel configuration options are consistent throughout the entire cluster.

Installing man pages

Now that all the kernels are installed, it's time to install the MOSIX man pages and user tools on every node in the cluster. First, let's tackle the man pages, since they're easier. To install the MOSIX man pages, untar the manuals.tar file included in the MOSIX-1.5.2 directory into your preferred man page directory, typically /usr/man or /usr/share/man:

```
# cd /usr/share/man # tar xvf /root/MOSIX-1.5.2/manuals.tar
```

Installing user tools

Now we're ready to install the MOSIX user tools. First, extract the user.tar tarball into its own directory:

```
# cd /root/MOSIX-1.5.2 # mkdir tools # cd tools # tar xvf
../user.tar
```

Before following these next steps, make sure that /usr/src/linux contains a MOSIX-enabled kernel source tree, or is a symlink that points to a MOSIX-enabled source tree. Then, use the following script to compile and install the MOSIX user tools:

```
#!/bin/sh for x in lib/moslib sbin/setpe sbin/tune bin/mosrun \
usr.bin/mon usr.bin/migrate usr.bin/mosctl do cd $x make && make
install cd ../.. done
```

Once you've installed the MOSIX user tools and man pages on every node in your cluster, all we need to do is get the MOSIX configuration files set up and then reboot.

Section 5. Final configuration steps

/etc/mosix.map

Let's get the configuration files set up now. First, we'll need to create a file called "/etc/mosix.map" that will describe all the nodes in our new cluster. When you're done editing this file, it should be copied verbatim to every node. All /etc/mosix.map files should be identical throughout the entire cluster.

The format of the mosix.map file is simple. Each line should contain a node number, the IP address or hostname of the node, and a "span", normally one. For example, this mosix.map file defines a three-node cluster:

```
1 192.168.1.1 1 2 192.168.1.2 1 3 192.168.1.3 1
```

/etc/mosix.map, continued

If you have a series of consecutive IP addresses like the example above, you can take advantage of the span argument and save some typing:

```
1 192.168.1.1 3
```

This mosix.map file, like the one above it, defines a three-node cluster consisting of IPs 196.168.1.1, 192.168.1.2, and 192.168.1.3. Since you're probably setting up a small cluster (at least initially), it's best to avoid using the span argument for the time being. Simply specify a span of "1" for each node, as in the first example.

MFS

Next, we need to create a mount point and /etc/fstab entry for MFS, the mosix filesystem. First, create a /mfs directory on each node:

```
# mkdir /mfs
```

Then, add an mfs entry to your /etc/fstab file:

```
mymfs /mfs mfs defaults 0 0
```

If you'd like to take advantage of DFSA and you've compiled DFSA support into your kernels, use this line in your fstab instead:

```
mymfs /mfs mfs dfsa=1 0 0
```

Understanding MFS

So, what does MFS do? Well, when we reboot and MOSIX is enabled, you'll be able to access filesystems throughout your cluster by perusing directories inside /mfs. For example, you can access the /etc/hosts file on node 2 by editing /mfs/2/etc/hosts, etc. MFS is extremely handy, and MOSIX also takes advantage of MFS to improve performance.

System startup

OK, we're almost done. Before we restart all the nodes in our new cluster, we need to add a couple of lines to each node's system startup scripts. First, add the following lines to your /etc/rc.d/init.d/local (or equivalent) startup file, so that the following commands are run at startup:

```
/sbin/setpe -W -f /etc/mosix.map touch /var/lock/subsys/mosix
```

These commands initialize the MOSIX node for use, and should execute *after* the local MFS filesystem has been mounted and *after* the network has been set up. Typically, if they are added to a "local" startup script, they'll execute at the right time.

Shutdown

Next, we need to ensure that the following lines are executed on shutdown, before the network is brought down:

```
echo 0 > /proc/mosix/admin/mospe rm -f /var/lock/subsys/mosix umount
/mfs
```

These commands remove the node from the cluster and also take care of unmounting the MFS filesystem. It's a good idea to unmount /fs manually, since some distributions' shutdown scripts will not recognize that /mfs is a networked filesystem and will try to unmount it at the wrong time.

One of the nice things about MOSIX is that it's perfectly safe to reboot a node while your cluster is in use. Any remote processes will simply migrate away before your system reboots. In the next section, we'll reboot and see MOSIX in action!

Section 6. MOSIX in action

We're ready!

Our cluster is now configured and ready for use. Now, go ahead and reboot all your systems, and when your systems come back up, MOSIX should be enabled and your cluster will be active. Once all the systems are rebooted, log in to one of the nodes and type:

```
# mon
```

You should be greeted with a console-based load meter. The active nodes in your cluster will be listed across the bottom of the graph by node number, and each node's load will be represented by a vertical bar.

Testing the cluster

Since you just rebooted all the machines, there probably isn't much happening on the nodes at the moment. So, let's do a little MOSIX test so that we can see process migration in action. To do this, we're going to create several very CPU-intensive processes on the local system, and then watch as MOSIX migrates these processes to the most efficient node. While keeping the "mon" program running, open at least two new shells, and run the following command in each of these shells:

```
awk 'BEGIN {for(i=0;i<10000;i++)for(j=0;j<10000;j++);}'
```

Understanding migration

With at least two of these commands running, head back to "mon". You should see the system load on your current node shoot up to 2 or more. But, after a few seconds, you should see the load on your current node drop and the load on another one or two of your MOSIX nodes increase. Congratulations; you've just witnessed MOSIX process migration in action! MOSIX detected that the new processes you created could run faster if some of them were migrated to other nodes in your cluster, and MOSIX did just that.

As far as the migrated processes are concerned, they're running on your local node, also called the process's "home" node. They have no idea that they are actually running on a remote CPU. In fact, the processes will still be listed in their home node's "ps" list, and they won't even show up in the remote node's "ps" list.

Cleaning up

Now that you've experienced your first taste of the power of MOSIX, you can go ahead and kill those CPU-hungry awk processes. In the next section, we're going run MOSIX through its paces and explore different ways that we can use MOSIX to increase performance and manage processes.

Section 7. Exploring MOSIX

Compiling, part 1

Since I do a lot of compiling on my machine, I was of course very curious how MOSIX could help compile times. So, almost immediately after getting MOSIX up and running, I tried to do a few test builds of some sources to see if I could see any performance improvement. In my tests, I used the "-j" option with make to tell make that it should run several gcc processes in parallel. For example, the "make -j 7" command will tell make that it should have 7 concurrent gcc processes (each compiling a separate source file) running at all times.

Compiling, part 2

Initially, my results were disappointing. Using "mon" to view the load on my my nodes, I saw that the load on node 1 remained consistently high, while the load on node 2 was 0.0 most of the time. MOSIX would only occasionally migrate a process from node 1 to node 2, and there was no significant compilation performance improvement.

After asking around a bit, I learned why my results were not that impressive. You see, gcc processes in general only exist for a few seconds; they're typically not around for long enough to make it worth MOSIX's while to migrate them to remote systems. Because the initial migration adds a decent amount of overhead, MOSIX was actually making the best choice by keeping most of the gcc processes running locally. So, while it looked like MOSIX wasn't doing its job, it was in fact being very smart by choosing not to migrate processes. Obviously, MOSIX's decision, while smart, wasn't very exciting, so I decided to look at another potential MOSIX application -- CD audio encoding.

A new test -- audio encoding!

For my next series of tests, I decided to see how MOSIX would fare with a completely different type of application. In my desire to learn more about how MOSIX worked, I created a little test. I extracted 13 tracks of digital audio from a music CD, and decided that I would compress these tracks using *FLAC*, a very good and free lossless audio encoder. By using FLAC, I can compress without compromising audio fidelity in any way.

Audio encoding, continued

This test is very different from the compilation test in a number of ways. For one, the audio compression processes have a relatively long lifetime, typically several minutes. Also, their execution speed is dependent primarily on CPU performance rather than IO. Due to these two factors, MOSIX should be better able to take advantage of the cluster's computing resources.

Introducing my cluster

But before we look at the results of my audio encoding tests, let me tell you you a bit about my cluster. My cluster consists of two nodes; node 1 is a 900Mhz AMD Athlon Thunderbird

system with 512Mb of RAM, and node 2 is a 650Mhz AMD Duron system with 384Mb of RAM. They are connected to one another using two 100Mbit fast ethernet cards running in full-duplex mode, thanks to a specially-wired CAT5 cable. While my cluster has the bare minimum number of nodes, I can use it to learn how to use MOSIX in an effective way. And as we'll see in a bit, even a 2-node cluster can do some amazing things.

The "baseline" test

To begin my test, I tried encoding all 13 tracks, one at a time . To do this, I logged into node 1, entered my CD track directory (which resides at /root/musictest on node 1's local filesystem) and typed the following mini-script at the command-line:

```
for x in track*.wav do flac -8 $x done
```

And the result? The system load on node 1 continually hovered at around 1.0 as the CD audio tracks were encoded one after another. The encoding completed in 11 minutes and 11 seconds. Throughout the entire process, the load on node 2 stayed at zero. Obviously, MOSIX didn't have a chance to kick in and migrate processes around, since I was only running one CPU-hungry process at a time, and node 1 had a more powerful CPU than node 2. This test provides us with a good baseline. This is the kind of performance we could expect from node 1 working alone, without MOSIX's, and thus node 2's, help.

MOSIX comes alive

For my next test, I decided to see what would happen if I ran all 13 encoding processes simultaneously. To do this, I used the following script:

```
for x in track*.wav do flac -8 $x & done
```

The addition of the trailing & caused 13 encoding processes to be launched immediately, rather than one after another. The result? Well, looking in my "mon" window, the system load on node 1 shot up to 12.0 almost immediately. But after a few seconds, the load on node 2 started rising, just as node 1's load started to fall. After about 15 seconds, the load on node 1 stabilized at 7, while node 2's load stabilized at 6. MOSIX had successfully (and automatically) migrated some processes to node 2!

This time, the encoding process completed in 6 minutes and 53 seconds, nearly half the time of the original 11 minutes and 11 seconds. This time is especially good considering that my 650Mhz node 2 isn't nearly as zippy as my 900Mhz node 1. In fact, if I had two equally-specced machines, it's possible that I would have seen an almost literal halving in CD audio encoding time. Impressive!

Analysis of MOSIX performance

By running all 13 tracks in concert, we allowed MOSIX to do what it does best. After MOSIX noticed that the load on node 1 was unusually high, it determined that migrating some processes would be the most efficient course of action. Because the processes had a relatively long lifetime, MOSIX had the opportunity to migrate them, and because the processes were CPU bound, their migration resulted in a dramatic improvement in encoding performance.

Example application -- University server

This particular test is a good example of MOSIX's suitability for handling a large number of resource-hungry processes. Consider a heavily-used University server that may have one hundred or more simultaneous login sessions and have an extremely high load. MOSIX is a truly ideal for these types of situations, since the high load and large number of processes means that MOSIX will have many opportunities to distribute the load among nodes in the cluster.

In a multi-user MOSIX server application, the server's performance can be improved by adding another node to the cluster. This is an entirely different paradigm than the typical approach, which involves scrapping the existing server and replacing it with a faster and pricier model. By using MOSIX, departments to continue to make use of their existing investments, resulting in huge cost savings.

The "runhome" test

Now, back to the CD encoding tests. In my next test, I decided to run all 13 encoding processes in parallel but use MOSIX's "runhome" command to prevent any processes from migrating from node 1 to any other node in the cluster. Here's the script that I used:

```
for x in track*.wav do runhome flac -8 $x & done
```

The results were as you might expect; encoding all the tracks in parallel really didn't help, since they all the processes had to fight for access to node 1's limited CPU timeslices. Encoding completed in 11 minutes and 13 seconds, just two seconds worse our linear baseline. The system load on node 1 stayed at a consistent 12.0, while node 2's load remained zero.

The "remote" test

Next, I wondered what would happen if I used MOSIX's "runon" command to force *all* encoding processes to execute remotely on node 2, forcing MOSIX to funnel data to and from node 1's filesystem and node 2's remote encoding processes. For this test, I used the "runon" command in the following script:

```
for x in track*.wav do runon 2 flac -8 $x & done
```

The results of this test were very interesting; system load on node 2 jumped to 17.0, which is something you probably expected. However, the system load on node 1 stayed at 0.0 throughout the entire encoding process. Node 1's load of 0.0 was quite impressive considering that node 1 needed to pump data to and from node 2 in order to allow all our encoding processes to run remotely. This would seem to indicate that at least for our CPU-bound processes, MOSIX has very little overhead. Encoding completed in 15 minutes and 57 seconds, significantly slower than our baseline but in line with node 1 and node 2's processor speed (900Mhz Thunderbird vs. 650Mhz Duron) differences.

Performance analysis of "remote" test

I found MOSIX's performance under the "remote" test to be an incredibly pleasant surprise.

Consider that while node 2 was chugging away running all those encoding processes, I had node 1 completely free to tackle other computing tasks. This was incredibly handy for me, since node 1 happens to be my desktop system; node 2 is my test box. This particular test clearly demonstrates how MOSIX can be used to offload CPU-intensive work to other machines, thus maximizing the interactive performance of your local workstation. And if my remote box(es) had combined processing power greater than that of my workstation, I could actually expect the audio encoding to be completed *faster* than if the processes ran locally, and all with a local workstation load of 0. Wow!

The "baseline remote" test, part 1

For my final CD encoding test, I decided to copy all of my CD tracks over to node 2 first, and then run the encoding processes on node 2 directly, using the "runhome" command to prevent them from migrating to node 1. By doing this, I hoped to determine exactly how much of the previous 15:57 result was due to MOSIX overhead, and how much was due to the relative difference in node 1 and node 2's processor speeds. To perform this test, I first took used MFS to copy the CD tracks from node 1 to node 2. On node 1, I typed:

```
# cd /root/musictest # mkdir /mfs/2/root/musictest # cp track*.wav
/mfs/2/root/musictest
```

MFS made it incredibly easy to copy my CD tracks from node 1's /root/musictest directory to node 2's /root/musictest directory. This is also a very good example of how you can use MFS to shuffle data from one node to another.

The "baseline remote" test, part 2

Once the CD tracks were on node 2, I ssh'd in and used the following script to encode the audio tracks:

```
for x in track*.wav do runhome flac -8 $x & done
```

Since the flac processes were started on node 2 with the "runhome" command, no processes were allowed to migrate to node 1. However, because the processes were running on node 2, *and* the data itself resided on node 2, MOSIX didn't need to do any work shuffling data back and forth.

Analyzing the results of "remote" tests

Now, what happened? System load on node 2 jumped to 16.9, and the encoding process completed in 15 minutes and 13 seconds. This helps us put the results in our previous test in perspective; in our previous example, we added only 4.8% overhead by running our processes on a remote node and relying on MOSIX to transparently shuffle data back and forth! In my opinion, that kind of overhead is incredibly reasonable.

Section 8. Resources and feedback

Resources

You've reached the end of my MOSIX tutorial; I hope you've had a fun time and have a good grasp of Linux clustering technologies and MOSIX in particular. Please take a look at the following resources to learn more about MOSIX:

To learn more about MOSIX installation, be sure to read the README included in the MOSIX source directory. In our test cluster installation, we only set up the bare minimum configuration to run MOSIX; further extensions to the MOSIX configuration are possible, allowing better fine-tuning of migration as well as better-tuned MOSIX system performance.

We're fortunate that the MOSIX team has assembled a wonderful set of MOSIX man pages. Be sure to type "man mosix"; in addition, take some time to read the man pages for setpe, mon, mosctl, migrate, mosrun, mosix.map, DFSA and MFS.

To learn more about MOSIX, visit the MOSIX site at *http://www.mosix.org*. At the MOSIX site, you'll find a FAQ as well as the mosix-user mailing list archives and subscription information. You'll also find links to various MOSIX papers and related research.

Your feedback

Thanks for reading my tutorial, and I hope that you continue to enjoy and benefit from MOSIX. We are truly fortunate that such a powerful and free technology is available for Linux systems.

I look forward to getting your feedback on this tutorial. Additionally, you are welcome to contact me directly at *drobbins@gentoo.org*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.