openMosix Clusters: Setup and Internals

Linux beginners shy away from when they hear the word Linux clusters. If you know how to compile a kernel, read on and you will be surprised at how simple it is to setup a cluster at home or at work to speed up your computational needs! Programmers, learn how openMosix achieves this capability by taking a peek into its internals.

Say Hello to openMosix

No matter how much computation power we seem to have, we never seem to be satisfied with it. It is much like money I guess. The more you have it, the more you want it. It is not that I am a gamer or anything, but I seem to want more computation power just for the heck of it. Well, it is just fun to have a powerful computer with a blazing CPU and loads of memory. In the quest for more CPU cycles, I began looking around for solutions. I looked at SMP and 64-bit solutions. SMP systems are very expensive. Linux does have great support for SMP systems. 64 bit system are powerful as well, but they are equally expensive. Fortunately, there are other solutions in the FOSS world. I wanted something good with the resources that I had at hand.

If you have a spare PC lying around gathering dust, go keep it ready. Lets make it part of a cluster and make it share the load, as much as it can, when it gets heavy on your main PC.

Working in the Technology Solutions department at Naturesoft in Chennai, I had at my disposal 3 P4 2.4 GHz systems with 512 MB RAM on each of them. It was time to make them work as one entity. It was time to build a cluster! I had heard a bit about openMosix and wanted to give it a try. There were other clustering solutions that came up when I googled, but openMosix seemed to be the most popular.

Not only was openMosix popular, as I later found out, it was trivial to setup and manage. I meet a lot of students in the process of Free Software advocacy and many of them had plans of using clusters in their projects. I am not aware of the reason, but they always seems to have considered a Beowulf based setup for their needs. Some seem to have read articles/HOWTOs on cluster building that require resources beyond a student's reach. When I revealed that at home I used a cluster of a Celeron 1.1 GHz machine with that of a 100MHz Pentium, many were instantly interested.

What is SSI?

You can build many types of clusters, but openMosix lets you build one particular type called an SSI cluster. SSI stands for Single System Image. Put simply, it means that in this type of configuration, the whole cluster looks like a single machine that has many CPUs and large amounts RAM. You do not need to get special hardware for this type of a cluster. You can string together normal PCs and build a cluster that presents to you the consolidated power of these systems. These PCs need to be networked, so they need a Network Interface Card(NIC) on each of the systems. For a cluster consisting of more than 2 systems, you will need a hub/switch to connect them together, but if you are just going to have two nodes, you can use a cross cable to link the two NICs together. There is no need for a switch or hub.

SSI and Beowulf Clusters: The Difference Explained

The advantage of openMosix is that you do not need to modify programs to make them run on the cluster. If you take a Beowulf cluster setup for instance, special, cluster aware

programs need to be written(with the help of libraries like MPI or PVM) to take advantage of the cluster. But you have to remember that Beowulf clusters serve a different purpose. They are able to run specially written parallel programs, whereas, the primary goal of an SSI cluster is load balancing and optimal use of resources. To make things clear, let me present an example here. Say you have to encode a .wav file into a .mp3 file. On Linux you can use the *lame* program to do this. As many of you might know, this is a time consuming operation. On a Beowulf cluster of say 10 nodes, you can modify the *lame* program to split the .wav file into 10 chunks and run parallel instances of *lame* on each of the nodes. Once done, the *lame* process can then consolidate the results and give you one single .mp3 file. Since the work is split and done among nodes simultaneously, it will take one tenth of the time compared to the time that it would take *lame* to run on a single PC. This is the kind of work Beowulf clusters are capable of doing, but you *need* to modify the *lame* program to take advantage of the Beowulf cluster.

An openMosix cluster on the other hand cannot be used for this kind of parallel computing. What you can however do, is run 10 instances of *lame* on the 10 nodes to convert 10 .wav files into 10 different .mp3 files. There is no need to modify the *lame* program in any way for this. And even if you start the 10 instances of *lame* on a single node on the cluster, openMosix will migrate them to other nodes and auto-balance the load. So, if you need to write programs that do a lot of computing on an openMosix cluster, all you need to do is write programs that spawn processes(using the fork( ) system call) to do different tasks. openMosix will automatically migrate them to auto-balance the cluster.

Program Migration in openMosix

When programs run, they need two crucial resources. One is RAM and the other CPU. In a cluster, each node(computer) has some finite amount of these resources. As discussed earlier, openMosix is an auto-balancing cluster. It makes sure that the load of running programs is optimally distributed across nodes in the cluster. Moreover, openMosix allows for a heterogeneous setup, where the nodes can be of different configurations. For example, you can have two P4, 2.4 GHz systems with 512MB RAM and one 700 Mhz Celeron system with 128 MB RAM. On an openMosix cluster, processes can be started on any of the nodes.

The node on which a process starts is called its Unique Home Node(UHN). At some point of time, if openMosix decides that this process id better off running on another node on the cluster, it will migrate it there. If a process on the cluster is CPU intensive, openMosix will try to migrate it to a node with relatively more CPU power. Each node is identified by a unique ID assigned to it. Each node is also assigned a number according to the resources it has. This number is used to calculate to which node processes need to be migrated when a migration decision has to be made by openMosix.

Setting up openMosix

openMosix is a kernel extension available as a patch to the 2.4 kernel series. An extension to the 2.6 series is in the works. Since there is no configuration involved, installing openMosix is just a matter of compiling the kernel and its userland programs. To install openMosix you first need to acquire the openMosix 2.4.26 patch, the openMosix utilities and the 2.4.26 kernel from the sites listed in the resources section of this article. **For your convenience all the necessary software has been provided in this month's LFY CD[Ed – Please note].** Just follow the mentioned steps:

Untar the kernel sources and patch it with openMosix

```
$ tar xvjf linux-2.4.26.tar.bz2
$ bunzip2 openMosix-2.4.26-1.bz2
$ mv openMosix-2.4.26-1 linux-2.4.26
$ patch -p1 < openMosix-2.4.26-1
```

Configure, compile and copy the kernel

```
$ make menuconfig
$ make dep bzImage modules
$ sudo make modules_install
$ sudo cp arch/i386/boot/bzImage /boot/kernel-2.4.26-om
$ sudo cp .config /boot/config-2.4.26-om
$ sudo cp System.map /boot/System.map-2.4.26-om
```

Here I assume that you know what drivers you need for your system and that you will compile them as modules or include it in the kernel directly.**[Ed – If LFY has any previous article on kernel compilation, please include a reference here]** When you run *make menuconfig*, you will see an openMosix sub-menu. The items there are fine in their default values. Do not change them! Don't forget to select the driver for your NIC, if you have added it to your PCs for experimentation with openMosix. You then need to edit your /etc/lilo.conf or /boot/grub/grub.conf file to add an entry for the openMosix enabled kernel. **Hint:** For creating the new openMosix kernel entry, you can copy the entry for your existing kernel and then modify the path to point to the new kernel.

Compilation of openmosix-tools requires that you make a symlink linux-openmosix, in /usr/src that points to the openMosix patched kernel. We shall create that first.

```
$ cd /usr/src
$ sudo ln -s linux-2.4.26 linux-openmosix
```

You can replace linux-2.4.26 with the actual path to the directory where you have untar'ed the Linux kernel on your system, if you have not extracted it into /usr/src.

Compiling other tools

```
$ tar xvzf openmosix-tools-0.3.6-2.tar.gz
$ cd openmosix-tools-0.3.6-2
$ ./configure
$ make && sudo make install
$ cd ..
$ tar xvzf openmosixview-1.5.tar.gz
$ cd openmosixview-1.5
$ ./configure
$ make && sudo make install
```

Start the Cluster!

You need to do the installation for all the machines that you want to participate on the cluster. Reboot your nodes with the new openMosix kernel. There are RPM packages available for various distributions, but I have used source based examples since they are common to all distributions. Please note that to compile openmosixview, you need to have the development support packages for Qt library installed. On many distributions the name might be qt-dev. If you have an old node and plan not ro install/run a GUI on it, you need not install the openmosixview package there. This package is just for monitoring/managing the cluster through a GUI. There is an openMosix startup script that needs to run when the system boots up. This is installed into the startup scripts directory. Use the command/utility for your distribution to make it run during system boot up. Else you need to run the script manually as root once you login.

```
# /etc/rc.d/init.d/openmosix start
```

To restart openMosix you can use

```
# /etc/rc.d/init.d/openmosix restart
```

Change the path according to your distribution. You are now running a cluster!

Cluster Management, GUI style

Now open openmosixview from any of the nodes. It shows you the load levels on the different nodes in the cluster, plus the overall load on the cluster as a whole.

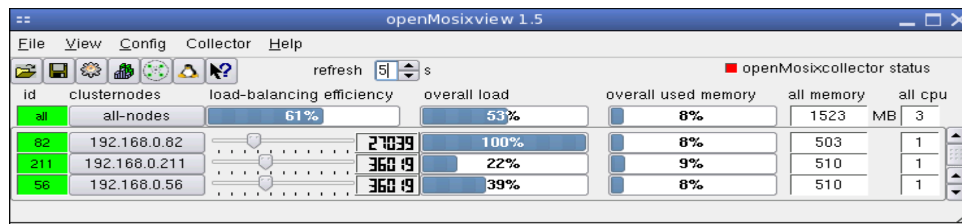

Figure-1 The openmosixview GUI

In the cluster that I'm running there are 3 nodes with the IDs 82, 211 and 56. openMosix has derived the ID from the machine's IP addresses automatically. You can also start other utility programs from openmosixview by clicking on the respective icons. Other programs that you can run are:

openmosixprocs – shows the processes running on the current machine and lets you manage them with activities like expelling remote processes running on the current node, forcing local processes to migrate to other nodes, etc.

openmosixmigmon – this program shows you graphically the location of processes in the node that it is running from. You can also drag processes running on the current node to other nodes!

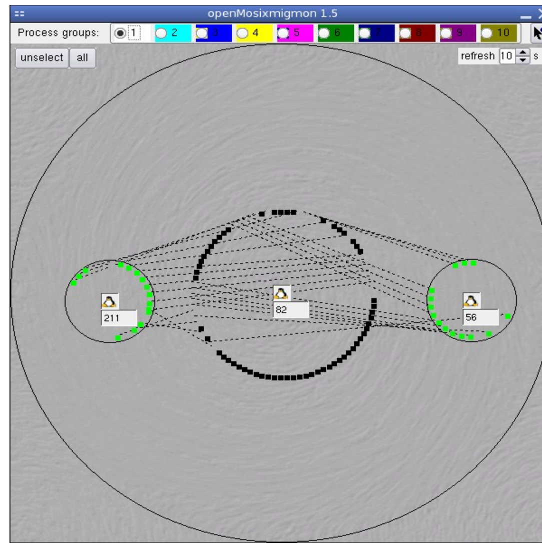

Figure-2 The openMosix migration monitor GUI

You can also run *mosmon* from the terminal to see the load levels of various nodes. The command *mosctl* lets you do administrative activities. See its man page for more information.

Lets put the cluster to work

Once you have a cluster, you need to use it. Well, you can just use it like you use your PC everyday. As you start processes on one node and its load increases beyond a point, processes start migrating out to other nodes. But please note that not all processes can migrate. Processes that use threads or shared memory cannot migrate from the UHN. But they will run fine. To see how openMosix migrates processes, we need to generate a lot of them. Lets now do just that. Go back to the kernel compilation directory. Keep the *openmosixmigmon* program running, so it will show you the migration process graphically. Run the following commands:

```
$ make clean
```

```
$ make -j10 dep bzImage
```

With the *-j10* option to the make command, you are telling it to compile up to 10 files in parallel before linking them all. This will create many processes. As you can see in the *opemmosixmigmon* GUI, processes belonging to the current node will migrate to other nodes. Hovering the mouse pointer over the tiny green/black squares that represent processes will give you their names and PIDs. You can also drag these tiny squares around nodes. This will initiate migration of the process.

The openMosix proc filesystem interface

openMosix uses the proc filesystem for the administrative interface. This makes it possible to use the command line to manage processes running on an openMosix cluster. Look at the following commands:

```
# echo "5" > /proc/4983/goto
# cat /proc/4983/where
```

The first command sends the process with PID 4983 to the node with ID 5. The second command tells you on which node the process with PID 4983 is running. Please note that most of the proc interface files for openMosix present information in a binary format. It is better to use commands like *mosctl* that in turn interact with the proc interface while providing human readable data to you.

How openMosix Does This

openMosix uses the UDP protocol and UDP ports in the range of 5000-5700 for inter-node communication. Since there is less overhead using UDP and since clusters are implemented on closed/private networks the chances of packets getting dropped are less. Unless, there might be serious problems like those associated with hardware. Inter-node communication needs to be quick and openMosix cannot afford the waiting time associated with TCP connection establishment and tear down. When a process has got to run on a different machine on the cluster, note that whole pages of memory containing code and data need to be migrated/moved across the network.

If you think this stuff is nifty, let me tell you that it is not very difficult to understand. Well, as usual, implementing it is a different story altogether! It is said that something thats well begun is half done. Similarly, UNIX is so well designed that adopting it for so many other things is easy. openMosix is no exception. The “everything is a file” model in UNIX has helped simplify things much. Well, I assume you know that Linux is a UNIX like operating system that takes full advantage of UNIX's design.

System Calls and openMosix

When a process is migrated from its home node, to another node, there is a small stub called the *Deputy* that represents that process on its Unique Home Node(UHN). On the remote node where it is actually being run, apart from the actual process itself, it is represented by a stub called the *Remote*. Remember that for many things, the remotely running process has to get back to its UHN. Actually, most of the system call that the remote process invokes, are executed on its UHN.

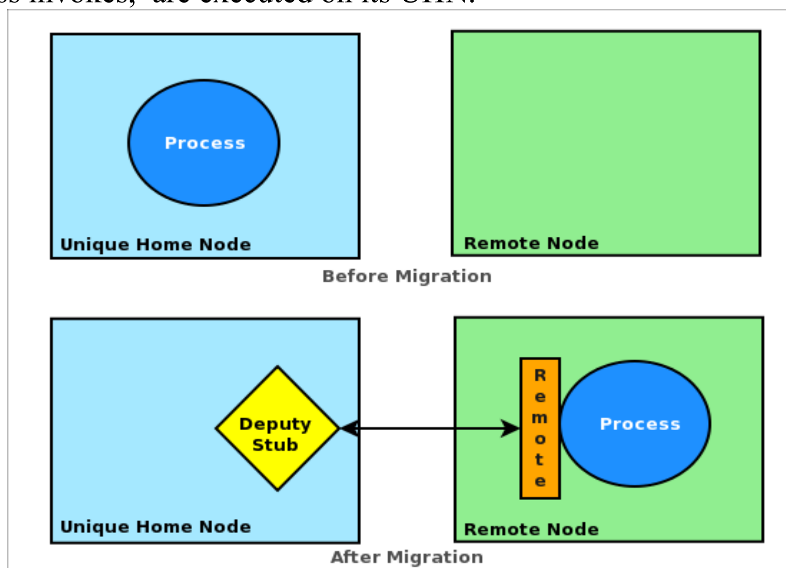

Figure-3 openMosix Design

Whenever a remote process makes a system call, the details of the call are passed on by the *remote* stub to the *deputy*. The *deputy* executes the call and sends back the result. Since most of the I/O in UNIX is done using the open()/read()/write()/ioctl()/close() system calls and there is nothing to be specially handled, life of the openMosix developers seems to have made a lot more easier. So there is nothing special to be done for each and every device that a remotely running process may be trying to access. Take for example the case with X applications. They communicate with the X Server using Internet or UNIX domain sockets. Socket I/O is done using just read() and write() system calls. Thus X applications that are migrated need no special treatment.

File I/O and openMosix

"A supercomputer is a device for turning compute-bound problems into I/O-bound problems." - ***Ken Batcher***

If there is a migrated process and it tries to do too much I/O all the file and system call data needs to be moved back and forth between its UHN and the node on which it is remotely executing. Remember that when the remote process tries to open a file, that file may not be present on the node on which it is running. Even if it is, it should not be accessing it, but rather it should be accessing the one that is present on its UHN. Thus when the process calls the *open* system call the *remote* stub passes this request to the *deputy* stub. The *deputy* stub calls the *open* system call as if it were the process itself. If the call is successful and a file handle is obtained, the handle is passed back as a return value to the *remote* stub which in turn provides the process with it.

An Example

The openMosix patch, when applied, modifies many files in the stock kernel. The openMosix specific code is kept in the 'hpc' directory under the kernel source tree. This directory contains code that deals with communication (comm.c), the deputy stub(deputy.c), the proc interface (hpcproc.c), the migration daemon(mig.c), the remote stub(remote.c), the system calls layer(syscalls.c), data cache management(ucache.c), etc.

Let us now walk through an example of a remote process making a system call and then see what happens in openMosix.

1. Program calls sys_open(“filename”,O_RDONLY)
2. remote_sys_open( ) from syscalls.c is called
3. remote_standard_system_call( ) from remote.c is called
4. remote_standard_system_call( ) packs the system call arguments and sends it over the network to the process's UHN. This is accomplished by calling these functions:

    construct_ucache_envelope( )

    comm_send( )
5. System call request received by deputy_main_loop( ) on UHN
6. Function deputy_syscall( ) is called which makes the actual system call on the UHN
7. Deputy sends return value through the deputy_reply( ) function which uses

comm_send( ) to send the return value to the remote stub

8. The remote stub receives the return value in the function remote_wait( ) and passes it on to the process.

Experienced readers might be still wondering about something amiss. When a remote process sends data related with a system call, like the file name and mode, in the case of sys_open, the file name is just a pointer. This pointer will not be valid on the UHN since the remote process is on another node altogether! This is where the ucache comes in. All the necessary data is packed along with the system call request before it is send to the deputy stub on the UHN. Data packing is done by the construct_ucache_envelope( ) function in step #4.

We are done

openMosix is a simple way to setup a cluster. For others, it is a good way of learning how clusters work. I hope this introduction would have kindled your hacker instincts enough to start tinkering with the openMosix kernel. Enjoy Free Software, Happy Hacking!

Resources

openMosix web site: http://openmosix.sourceforge.net/

openMosix 2.6 port: http://openmosix.snarc.org/

openMosix IRC Channel : server: Freenode.net channel:#openmosix (Many developers sleep here)

The Beowulf Project: http://www.beowulf.org/

The OpenSSI project: http://www.openssi.org/

openmosixview web site: http://www.openmosixview.com/

About the Author:
Shuveb Hussain enjoys programming Linux and loves the freedom offered by Free Software. The author blogs at http://binarykarma.org.
You can reach him at shuveb at binarykarma dot org. *This article first appeared in Linux For You (http://lfymag.com).*