CSCNSI Bootcamp Guides 2019*

J. Lowell Wofford[†], LANL HPC-DES
Cory Lueninghoener[‡], LANL HPC-DES

July 24, 2019

Contents

*This work is part of the Los Alamos National Laboratory, *Computer Systems, Cluster and Netowrking Summer Institute* (CSCNSI) as part of the *Information Science and Technology Institute* (ISTI). This document has been authorized for full release: LA-UR-19-26998.

[†]lowell@lanl.gov

[‡]cluening@lanl.gov

Exploring Linux Part 1 - Linux Fundamentals

Prerequisites

At this point:

1. your master node should be cabled;
2. your master node should have a fresh install of CentOS;
3. your master node should have a special account called admin;
4. your master node should be configured to live on the 10.0.52.xxx;

Part 1: Access & adding users

We added an admin user to the system. We did this because, by default, ssh will not allow root to login directly; however, we want to setup individual users for the master.

Let's start by logging into the master. Open at terminal on your Mac laptop while connected to the wired network. Now login using: ssh admin@10.0.52.xxx. Enter the password when prompted. You should be given a bash prompt:

```
lowell$ ssh admin@10.0.52.xxx
Last login: Sun Jun  2 18:53:37 2019 from 10.0.52.52
[admin@localhost ~]$
```

We will need to become root to add our users. It should have been configured at install time that the admin user can have full access to the sudo command. sudo allows an ordinary user to use their own password to perform actions as another user. We can check what sudo privileges we've been given with (using the password of admin, not root):

```
[admin@localhost ~]$ sudo -l
[sudo] password for admin:
Matching Defaults entries for admin on te-master:
    !visiblepw, always_set_home, match_group_by_gid,
 ↪  always_query_group_plugin, env_reset, env_keep="COLORS DISPLAY HOSTNAME
 ↪  HISTSIZE
    KDEDIR LS_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME LANG
 ↪  LC_ADDRESS LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION
    LC_MEASUREMENT LC_MESSAGES", env_keep+="LC_MONETARY LC_NAME LC_NUMERIC
 ↪  LC_PAPER LC_TELEPHONE", env_keep+="LC_TIME LC_ALL LANGUAGE
    LINGUAS _XKB_CHARSET XAUTHORITY",
 ↪  secure_path=/sbin\:/bin\:/usr/sbin\:/usr/bin

User admin may run the following commands on te-master:
    (ALL) ALL
```

The last line of this output tells us that admin can run ALL commands as ALL users using sudo.

We can now use this account to run a command as root:

```
[admin@localhost ~]$ sudo whoami
root
```

We can launch a new interactive session as root with the the −i option:

```
[admin@localhost ~]$ sudo -i
[root@localhost ~]#
```

Let's get things started by setting the hostname of our master. This guide will use the hostname te−master. The systemd way of doing this is:

```
# hostnamectl set-hostname te-master
```

Now that we are root, we have the power to add new users for each member of the group. We can do this with the useradd command, then set a password with the passwd command. Let's suppose our first user login is jane (just add one user for now).

```
# useradd jane
# passwd jane
Changing password for user jane.
```

```
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
```

Let's look at what this created. These are the files that contain user information. Let's look at what we created (if you're unfamiliar, grep can find and print lines in a file that have a matching string):

```
[root@te-master ~]# grep jane /etc/passwd
jane:x:1003:1003::/home/jane:/bin/bash
[root@te-master ~]# grep jane /etc/group
jane:x:1003:
[root@te-master ~]# grep jane /etc/shadow
jane:$6$1i1DFQ0F$/6w6OsZjegp.fWspIWRUHzsZMalvh6NUJ2UKnKJqFhiqBACcvM87bawsF⌟
 ↪  RL0dj9bgU1kZV8SfDniH9le.6upT/:18051:0:99999:7:::
```

/etc/passwd contains the user information; /etc/group contains group information (jane's default group is named jane as well); /etc/shadow contains account age and password hash information.

We want jane to have sudo power. Let's see if she does:

```
# sudo -l -U jane
User jane is not allowed to run sudo on te-master.
```

Nope. jane doesn't have sudo. Why is this? We can inspect the sudo configuration. We could just open the file: /etc/sudoers, but there's a better way. Instead, run sudoedit. sudoedit will open whatever editor is specified by the EDITOR environment variable. By default, this is vi (see the supplied vi cheatsheet). It has the advantage that it will check the syntax of the file before saving it. This is a *very* good idea. A mangled /etc/sudoers file could block access to everyone.

```
# sudoedit /etc/sudoers
...
    105 ## Allows people in group wheel to run all commands
    106 %wheel  ALL=(ALL)       ALL
    107
...
```

Somewhere around line 106 we see this specification. It means that members of the group named wheel can use all sudo commands as all users. So, we just need to add jane to the wheel group. Let's do this by editing the /etc/group file. Like sudoedit, there's a safe way to do this:

```
# vigr
...
wheel:x:10:lowell,jane
...
```

We add jane to the line starting with wheel, after the last colon. This is a coma separated list.

Let's take a look at jane now:

```
# id jane
uid=1003(jane) gid=1003(jane) groups=1003(jane),10(wheel)
```

Great! jane is now in the wheel group. Let's check sudo again:

```
# sudo -l -U jane
Matching Defaults entries for jane on te-master:
    !visiblepw, always_set_home, match_group_by_gid,
 ↪  always_query_group_plugin, env_reset, env_keep="COLORS DISPLAY HOSTNAME
 ↪  HISTSIZE
    KDEDIR LS_COLORS", env_keep+="MAIL PS1 PS2 QTDIR USERNAME LANG
 ↪  LC_ADDRESS LC_CTYPE", env_keep+="LC_COLLATE LC_IDENTIFICATION
    LC_MEASUREMENT LC_MESSAGES", env_keep+="LC_MONETARY LC_NAME LC_NUMERIC
 ↪  LC_PAPER LC_TELEPHONE", env_keep+="LC_TIME LC_ALL LANGUAGE
    LINGUAS _XKB_CHARSET XAUTHORITY",
 ↪  secure_path=/sbin\:/bin\:/usr/sbin\:/usr/bin

User jane may run the following commands on te-master:
    (ALL) ALL
```

Ok, now that jane has access, we let's add the other users. Now that we know aboug wheel, we can do that in one step.

```
# useradd -G wheel <member_login>
# passwd <member_login>
```

That's it! We should now have all members added as users. Each member should now login as their own user and try sudo:

```
$ ssh jane@10.0.52.xxx
[jane@te-master] $ sudo -i
[root@te-master] #
```

Part 2: Yum, RPM & repos

For the moment we will need to choose one person to perform commands on our master. We will work up to a more collaborative way to do things, but we first need to be able to install some new software.

Before we do anything else, we need to setup some custom repositories. We will be using local mirrors of the repositories. A file is available that contains all of this information. We can grab it like this:

```
[root@te-master ~]# curl -O http://10.0.52.146/repo/yum.repos.d.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
 ↪  Current
                                 Dload  Upload   Total   Spent    Left
 ↪  Speed
100  2773  100  2773    0     0   502k      0 --:--:-- --:--:-- --:--:--
 ↪  541k
[root@te-master ~]# ls
yum.repos.d.tar.gz
```

Now we can extract it:

```
[root@te-master ~]# tar zxvf yum.repos.d.tar.gz
yum.repos.d/
yum.repos.d/CentOS-CR.repo
yum.repos.d/CentOS-Debuginfo.repo
yum.repos.d/CentOS-Media.repo
yum.repos.d/CentOS-Sources.repo
yum.repos.d/CentOS-Vault.repo
yum.repos.d/CentOS-fasttrack.repo
yum.repos.d/CentOS-Base.repo
yum.repos.d/CentOS-Base.repo.rpmnew
yum.repos.d/epel.repo
yum.repos.d/OpenHPC.repo
yum.repos.d/OpenHPC-updates.repo
yum.repos.d/mlnx.repo
```

Repository specifications live in /etc/yum.repos.d (take a look!). We can copy these files into place to use our local repos:

```
# /bin/cp -avf yum.repos.d/* /etc/yum.repos.d/
‘yum.repos.d/CentOS-Base.repo’ -> ‘/etc/yum.repos.d/CentOS-Base.repo’
‘yum.repos.d/CentOS-Base.repo.rpmnew’ ->
 ↪  ‘/etc/yum.repos.d/CentOS-Base.repo.rpmnew’
‘yum.repos.d/CentOS-CR.repo’ -> ‘/etc/yum.repos.d/CentOS-CR.repo’
‘yum.repos.d/CentOS-Debuginfo.repo’ ->
 ↪  ‘/etc/yum.repos.d/CentOS-Debuginfo.repo’
‘yum.repos.d/CentOS-fasttrack.repo’ ->
 ↪  ‘/etc/yum.repos.d/CentOS-fasttrack.repo’
‘yum.repos.d/CentOS-Media.repo’ -> ‘/etc/yum.repos.d/CentOS-Media.repo’
‘yum.repos.d/CentOS-Sources.repo’ -> ‘/etc/yum.repos.d/CentOS-Sources.repo’
‘yum.repos.d/CentOS-Vault.repo’ -> ‘/etc/yum.repos.d/CentOS-Vault.repo’
‘yum.repos.d/epel.repo’ -> ‘/etc/yum.repos.d/epel.repo’
‘yum.repos.d/mlnx.repo’ -> ‘/etc/yum.repos.d/mlnx.repo’
‘yum.repos.d/OpenHPC.repo’ -> ‘/etc/yum.repos.d/OpenHPC.repo’
‘yum.repos.d/OpenHPC-updates.repo’ ->
 ↪  ‘/etc/yum.repos.d/OpenHPC-updates.repo’
```

Note: we used /bin/cp instad of cp to avoid having to verify that we want to overwrite files.

Let's test this out and install a useful package. vim is the "improved" version of vi and has useful features like syntax highlighting. Let's install it:

```
# yum -y install vim
...
Installed:
  vim-enhanced.x86_64 2:7.4.160-5.el7

Dependency Installed:
  ...
  perl-threads-shared.x86_64 0:1.43-6.el7          vim-common.x86_64
 ↪  2:7.4.160-5.el7         vim-filesystem.x86_64 2:7.4.160-5.el7

Complete!
```

Now let's look at one of those repo files: vim /etc/yum.repos.d/CentOS−base.repo. You should see some nice syntax highlighting.

Here's another one that will be useful later: yum −y install telnet

Let's see what we installed. We can list the files in an installed package with the rpm command:

```
# rpm -ql telnet
/usr/bin/telnet
/usr/share/doc/telnet-0.17
/usr/share/doc/telnet-0.17/README
/usr/share/man/man1/telnet.1.gz
```

We can list all of the RPMs installed on the system with:

```
# rpm -qa
...
audit-2.8.4-4.el7.x86_64
kernel-3.10.0-957.12.2.el7.x86_64
irqbalance-1.0.7-11.el7.x86_64
aic94xx-firmware-30-6.el7.noarch
microcode_ctl-2.1-47.2.el7_6.x86_64
parted-3.1-29.el7.x86_64
kernel-tools-3.10.0-957.12.2.el7.x86_64
iprutils-2.4.16.1-1.el7.x86_64
sudo-1.8.23-3.el7.x86_64
```

We can do that with yum too:

```
# yum list installed
...
xz.x86_64                      5.2.2-1.el7                    @base
```

```
xz-libs.x86_64                   5.2.2-1.el7
 ↪  @base/$releasever
yum.noarch                       3.4.3-161.el7.centos
 ↪  @base/$releasever
yum-metadata-parser.x86_64       1.1.4-10.el7
 ↪  @base/$releasever
yum-plugin-fastestmirror.noarch 1.1.31-50.el7
 ↪  @base/$releasever
zlib.x86_64                      1.2.7-18.el7
 ↪  @base/$releasever
```

We can look at other lists like yum list available to see available packages.

If we needed to update software, we could use:

```
# yum update
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
No packages marked for update
```

. . . but we may not currently have any updates.

Suppose we want to know what package provided us the /bin/ls command:

```
# rpm -q --whatprovides /bin/ls
coreutils-8.22-23.el7.x86_64
```

But this only works for things that are installed. With yum, we can find out what package would provide the fortune command even if it's not installed. We can run:

```
# yum provides '*bin/fortune'
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
epel/filelists_db
 ↪                                        |  16 MB  00:00:00
fortune-mod-1.99.1-17.el7.x86_64 : A program which will display a fortune
Repo        : epel
Matched from:
Filename    : /usr/bin/fortune
```

Part 3: Sharing a session with tmux

Let's install a tool that we will be using to collaborate:

```
yum -y install tmux
...
```

Now, one member of the team should run the tmux command as root.

The screen should now have a green bar at the bottom with some text in it. tmux is a handy utility that allows you to do quite a few useful things. Most tmux commands are of the form <Ctrl−b>−<command>.

1. You can create a new terminal in one login <Ctrl−b−c>. To switch to the next window, <Ctrl−b−n>, and the previous <Ctrl−b−p>.
2. You an split your screen into two terminals: <Ctrl−b−">. Switch to the top screen <Ctrl−b−UP>.

There's a lot more you can do with tmux. See the included cheatsheet.

There's one specific thing we want to do with screen though. The other two users should login from their systems, then become root. Each can then run:

```
# tmux at
```

This will "attach" everyone to the same tmux session so that what one person types, the others see.

Take a little time to play around with tmux using the cheatsheet.

Part 4: files and permissions

It might be useful to have a directory that all three uses can work together in without using root. Let's set that up.

First, to share files, there will need to be a common group. Your group can be named anything you like. We will use teachers. To create the group: groupadd teachers

We can add jane to teachers with the usermod command. If we want to get fancy, though, we can add all of the users to the group in one line. Let's say the other two users are francis and lucy. The bash shell provides the ability to write for loops for this kind of thing:

```
# for u in jane francis lucy; do usermod -a -G teachers $u; id $u; done
uid=1004(jane) gid=1005(jane) groups=1005(jane),1002(teachers)
uid=1002(francis) gid=1003(francis) groups=1003(francis),1002(teachers)
uid=1003(lucy) gid=1004(lucy) groups=1004(lucy),1002(teachers)
```

This added the users to the group, then showed their membership to verify.

We need a place to put our shared directory. Let's put it at /home/share/teachers. The directory /home/share doesn't exist. We can make both /home/share and /home/share/teachers in one command:

```
mkdir -p /home/share/teachers
```

Let's check the permissions of our teachers directory:

```
# ls -ld /home/share/teachers
drwxr-xr-x 2 root root 6 Jun  3 22:08 /home/share/teachers
```

This isn't quite what we want. We want anyone in teachers to be able to write here. We also don't want anyone else to be able to look here. We need to set the group of the directory. Let's also make jane the owner:

```
# chown jane:teachers /home/share/teachers
[root@te-master ~]# ls -ld /home/share/teachers
drwxr-xr-x 2 jane teachers 6 Jun  3 22:08 /home/share/teachers
```

Better, but we want the group to be able to write:

```
# chmod 775 /home/share/teachers
[root@te-master ~]# ls -ld /home/share/teachers
drwxrwxr-x 2 jane teachers 6 Jun  3 22:08 /home/share/teachers
```

But we also don't want anyone else to have access:

```
# chmod o-rxw /home/share/teachers
[root@te-master ~]# ls -ld /home/share/teachers
drwxrwx--- 2 jane teachers 6 Jun  3 22:08 /home/share/teachers
```

As your user (not root), create a file in the directory. We can create an empty file with the touch command (on an existing file, this would update the modify time):

```
[jane@te-master ~]$ cd /home/share/teachers
[jane@te-master teachers]$ touch afile
[jane@te-master teachers]$ ls -l
total 0
-rw-rw-r-- 1 jane jane 0 Jun  3 22:15 afile
```

Once each user has done this, how would we find out which one was created last?

```
[jane@te-master teachers]$ ls -ltr
total 0
-rw-rw-r-- 1 jane    jane    0 Jun  3 22:15 afile
-rw-rw-r-- 1 francis francis 0 Jun  3 22:16 bfile
-rw-rw-r-- 1 lucy    lucy    0 Jun  3 22:17 cfile
```

The −t option sorts by time. The −r option gives reverse order. This puts the most recently modified at the bottom.

This may not be quite what we want though. We want files made here to retain the teachers group permissions. We can set the setgid bit on the directory for this.

```
[jane@te-master teachers]$ chmod g+s /home/share/teachers/
[jane@te-master teachers]$ ls -ld /home/share/teachers
drwxrws--- 2 jane teachers 45 Jun  3 22:21 /home/share/teachers
[jane@te-master teachers]$ touch test
[jane@te-master teachers]$ ls -l
total 0
-rw-rw-r-- 1 jane    jane     0 Jun  3 22:15 afile
-rw-rw-r-- 1 francis francis  0 Jun  3 22:16 bfile
```

```
-rw-rw-r-- 1 lucy    lucy     0 Jun  3 22:17 cfile
-rw-rw-r-- 1 jane    teachers 0 Jun  3 22:23 test
```

That's better!

Part 5: Shell tricks

We have been using bash as our shell, but we haven't explored its full potential very well.

Here are some handy things we can do:

Redirection and pipes

We can redirect stdout (where normal output goes) to a file:

Get all users with UID > 1000 and put them in the file high−uid.txt:

```
[root@localhost ~]# awk -F: '$3>1000{print}' /etc/passwd
jane:x:1001:1001::/home/jane:/bin/bash
francis:x:1002:1002::/home/francis:/bin/bash
lucy:x:1003:1003::/home/lucy:/bin/bash
[root@localhost ~]# awk -F: '$3>1000{print}' /etc/passwd > high-uid.txt
[root@localhost ~]# cat high-uid.txt
jane:x:1001:1001::/home/jane:/bin/bash
francis:x:1002:1002::/home/francis:/bin/bash
lucy:x:1003:1003::/home/lucy:/bin/bash
```

Here we used awk. Awk is a powerful parser that specializes in any thing that can be thought of as a grid of columns and rows. awk is actually a fully featured programming language in many ways.

Error messages are typically printed on stderr not stdout. We can redirect stderr to a file with <cmd> 2> <file>.

```
[root@localhost ~]# ls blah > out 2> err
[root@localhost ~]# cat out
[root@localhost ~]# cat err
ls: cannot access blah: No such file or directory
```

Or, we could combine them into one file:

```
[root@localhost ~]# ls blah > out 2>&1
[root@localhost ~]# cat out
ls: cannot access blah: No such file or directory
```

You can think of this as redirecting stderr into stdout. Note that this redirection happens after the redirection out.

Lots of commands can operate on data streamed to their standard input. We can pipe the output of one command to another. Suppose we want to see only the non-comment and non-empty lines in /etc/sudoers:

```
[root@localhost ~]# grep -vE '^#|^$' /etc/sudoers
Defaults   !visiblepw
Defaults    always_set_home
Defaults    match_group_by_gid
Defaults    always_query_group_plugin
Defaults    env_reset
Defaults    env_keep =  "COLORS DISPLAY HOSTNAME HISTSIZE KDEDIR LS_COLORS"
Defaults    env_keep += "MAIL PS1 PS2 QTDIR USERNAME LANG LC_ADDRESS
 ↪  LC_CTYPE"
Defaults    env_keep += "LC_COLLATE LC_IDENTIFICATION LC_MEASUREMENT
 ↪  LC_MESSAGES"
Defaults    env_keep += "LC_MONETARY LC_NAME LC_NUMERIC LC_PAPER
 ↪  LC_TELEPHONE"
Defaults    env_keep += "LC_TIME LC_ALL LANGUAGE LINGUAS _XKB_CHARSET
 ↪  XAUTHORITY"
Defaults    secure_path = /sbin:/bin:/usr/sbin:/usr/bin
root    ALL=(ALL)       ALL
%wheel  ALL=(ALL)       ALL
```

But, what if we just want to know how many lines there are?

```
[root@localhost ~]# grep -vE '^#|^$' /etc/sudoers | wc -l
13
```

Environment variables

Environment variables can control or track many things about your login session. Some of the special ones control bash itself, while others might be used for specific programs.

One helpful one is $HOME. This is your home directory. You can use it like this;

```
[jane@localhost ~]$ ls -ld $HOME
drwx------. 2 jane jane 83 Jun  4 05:17 /home/jane
```

Suppose you put some executables in $HOME/scripts, and you want them to be found automatically. You need to update your path:

```
[jane@localhost ~]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/jane/.local/bi⌋
 ↪  n:/home/jane/bin
[jane@localhost ~]$ export PATH=$PATH:$HOME/scripts
[jane@localhost ~]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/home/jane/.local/bi⌋
 ↪  n:/home/jane/bin:/home/jane/scripts
```

You can change your prompt by setting the PS1 variable.

```
[jane@localhost ~]$ export PS1="[nifty] $PS1"
[nifty] [jane@localhost ~]$
```

If you want this to persist, add it as a line to your $HOME/.bash_profile. To reload that in the current session, you can do that in one of two equivalent ways:

```
$ source $HOME/.bash_profile
$ . $HOME/.bash_profile
```

Working with history

What were the last 3 commands we ran?

```
# history | tail -n 3
  168  grep -vE '^#|^$' /etc/sudoers
  169  grep -vE '^#|^$' /etc/sudoers | wc -l
  170  history | tail -n 3
```

What was that crazy awk command again?

```
[root@localhost ~]# history | grep awk
  178  awk -F: '$3>1000{print}' /etc/passwd
  179  awk -F: '$3>1000{print}' /etc/passwd > high-uid.txt
  171  history | grep awk
```

I just want to re-run that.

```
[root@localhost ~]# !178
awk -F: '$3>1000{print}' /etc/passwd
jane:x:1001:1001::/home/jane:/bin/bash
francis:x:1002:1002::/home/francis:/bin/bash
lucy:x:1003:1003::/home/lucy:/bin/bash
```

If you want to search up through your history, you can use <Ctrl−r>, then start typing your search. When you find the match you want, hit enter (or <Ctrl−u> if you want to edit it).

```
(reverse-i-search)`rpm': rpm -q --whatprovides /bin/ls
```

When all else fails, `man`

So some one shows you this command that recursively prints the line count of every file less than a day old in a directory:

```
[root@localhost ~]# find . -type f -ctime -1 -exec wc -l {} \;
183 ./.bash_history
11 ./.lesshst
10 ./yum.repos.d.tar.gz
```

```
28 ./yum.repos.d/CentOS-CR.repo
21 ./yum.repos.d/CentOS-Debuginfo.repo
22 ./yum.repos.d/CentOS-Media.repo
42 ./yum.repos.d/CentOS-Sources.repo
...
```

Wait, how does −exec work in find again? Find out!

```
# man find
```

What man pages are availabe that mention vim?

```
# man -k systemd
...(lots and lots and lots and lots)
systemd.mount (5)    - Mount unit configuration
systemd.path (5)     - Path unit configuration
systemd.preset (5)   - Service enablement presets
systemd.resource-control (5) - Resource control unit settings
systemd.scope (5)    - Scope unit configuration
systemd.service (5)  - Service unit configuration
systemd.slice (5)    - Slice unit configuration
systemd.snapshot (5) - Snapshot unit configuration
systemd.socket (5)   - Socket unit configuration
systemd.special (7)  - Special systemd units
systemd.swap (5)     - Swap unit configuration
systemd.target (5)   - Target unit configuration
systemd.time (7)     - Time and date specifications
systemd.timer (5)    - Timer unit configuration
systemd.unit (5)     - Unit configuration
```

Load a specific man page number:

```
man 5 systemd.target
```

Linux Inspection

This guide has a lot of commands. For each command, it describes the command, shows an example of it running, and describes what the output means. As you walk through it, you should run each command on our own system to see what the output looks like and confirm that the output makes sense for your system.

Part 1: System Status

There are a number of commands that are useful for finding the general status of a system. These can be helpful for debugging why a system is misbehaving (are we out of memory? do we have free disk space?), seeing how loaded a system is (how may people are logged in right now? how many processes are running?), and lots of other basic discovery tasks.

uptime

The uptime command gives a super-simple overview of what's going on on the system right now:

```
[root@te-master ~]# uptime
 20:58:59 up  6:56,  1 user,  load average: 0.00, 0.01, 0.05
```

The output of uptime is a single line with four fields:

- The current time. In this case, it is almost 9:00pm (20:58:59)
- How long the system has been running since last reboot. In this case, the system was powered on six hours and 56 minutes ago (up 6:56)
- The number of users that are currently logged in. In this case, there is currently one user logged in. (1 user)
- The load on the system. The "load" is, over a given time, the average number of processes that are running or able to run. It is presented as the averages over the last one, five, and 15 minutes (load average: 0.00, 0.01, 0.05)

This command is useful for getting a very simple view of the system: how long it's been up, how loaded with people it is, and how loaded with processes it is. If you suspect something is wrong on a system, it can be a place to start narrowing down where to look next.

uname

The uname command, short for "Unix name", tells you what operating system kernel is currently running on your system. Running it with the −a option gives more information than running the command alone:

```
[root@te-master ~]# uname
Linux
[root@te-master ~]# uname -a
Linux te-master 3.10.0-957.12.2.el7.x86_64 #1 SMP Tue May 14 21:24:32 UTC
 ↪  2019 x86_64 x86_64 x86_64 GNU/Linux
```

In this case, we're running Linux on a system named te−master. The release of the Linux kernel being run is 3.10.0−957.12.2.el7 .x86_64, and its version has been set at compile time to have information about when it was compiled: #1 SMP Tue May 14 21:24:32 UTC 2019. The name of the machine hardware the kernel is running on is x86_64 (the first one), its processor is x86_64 (the second one), and its hardware platform is x86_64 (the last one). The operating system that the kernel is powering is called GNU/Linux.

This command gives an overview of what kind OS and hardware you are dealing with when you log into a system. You may find that a particular system is running a really old kernel, that it is running with an ARM processor, or that you are actually running a different OS than Linux. For example, the output of this same command on MacOS looks like this:

```
Darwin pn1804004.lanl.gov 18.5.0 Darwin Kernel Version 18.5.0: Mon Mar 11
 ↪  20:40:32 PDT 2019; root:xnu-4903.251.3~3/RELEASE_X86_64 x86_64
```

Note that the MacOS output is in a slightly different format, so you won't be able to directly map the Linux command's fields to this one.

df, du, and free

The df, du, and free commands are used to get information on storage utilization (df and du) and memory utilization (free) on a system. By default, they will each report in bytes or blocks, but they can each take a −h option that will make them report their numbers in human-readable format. Looking at them in turn:

df ("disk free")

```
[root@te-master ~]# df -h
Filesystem          Size  Used Avail Use% Mounted on
/dev/sda2           400G  9.9G  390G   3% /
devtmpfs            9.8G     0  9.8G   0% /dev
tmpfs               9.9G     0  9.9G   0% /dev/shm
tmpfs               9.9G  8.6M  9.8G   1% /run
tmpfs               9.9G     0  9.9G   0% /sys/fs/cgroup
172.16.1.253:/data  3.2T   51G  3.2T   2% /data
tmpfs               2.0G     0  2.0G   0% /run/user/0
```

The df command lists all of the filesystems on a system, how big they are, what their utilization is, and where they are mounted. In this case, we have lots of space everywhere.

du ("disk used")

```
[root@te-master /]# du -h
93M ./var/lib/rpm
0 ./var/lib/yum/repos/x86_64/$releasever/base
0 ./var/lib/yum/repos/x86_64/$releasever/extras
0 ./var/lib/yum/repos/x86_64/$releasever/updates
0 ./var/lib/yum/repos/x86_64/$releasever
...
4.0K ./data/ansible/.git/logs/refs/heads
4.0K ./data/ansible/.git/logs/refs
8.0K ./data/ansible/.git/logs
944K ./data/ansible/.git
1.5M ./data/ansible
```

```
50G ./data
60G .
```

The du command walks through an entire directory tree (by default, starting in your current directory) and calculates how much storage space is taken up by that directory and its subdirectories. In this case, it ran in the root directory (/), and the output was too long to fully include here. The last line tells us that there is 60GB worth of data in / and all of its subdirectories.

free ("free memory")

```
[root@te-master ~]# free -h
              total        used        free      shared  buff/cache
 ↪  available
Mem:            19G        596M         18G        8.6M        630M
 ↪  18G
Swap:            0B          0B          0B
```

The free command tells you how much RAM is available and being used on the system. In this case, the system has a total of 19GB of RAM, of which 596MB are being used. This leaves about 18GB (rounding!) of memory available. The shared field is somewhat misleading: it tells us that 8.6MB is used by RAM-based tmpfs filesystems. You can see that same number in the df output earlier. The buff/cache field is memory used by the kernel for caching, and the available field is similar to the free field, but takes into account the memory being used by the kernel.

These three commands each give insight into the current state of consumable components of your system. They can be used to monitor the usage of these components and take actions if needed: delete files, add more storage, kill processes, add more memory, etc.

w and last

The w and last commands provide information about who is currently or has recently been using your system.

w ("who")

```
[root@te-master ~]# w
 22:50:49 up 2 days,  8:48,  3 users,  load average: 0.00, 0.01, 0.05
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
root     pts/0    te-hyperv        21:31    1.00s  0.11s  0.11s -bash
lowell   pts/2    te-hyperv        22:29   21:29   0.02s  0.02s -bash
cluening pts/3    te-hyperv        22:50    4.00s  0.02s  0.02s -bash
```

The w command, which is an abreviation for who, starts out by providing the same information that uptime provides. It then lists all of the users that are currently logged in to the system, where they are logged in from, when they logged in, how busy they have been, and what they are running.

last ("last users")

```
[root@te-master ~]# last
root     pts/0        te-hyperv        Thu May 30 20:11   still logged in
root     pts/2        te-hyperv        Thu May 30 14:46 - 14:59  (00:13)
root     pts/2        te-hyperv        Thu May 30 14:44 - 14:45  (00:01)
...
root     tty1                          Thu May 23 17:48 - 17:48  (00:00)
reboot   system boot  3.10.0-957.12.2. Thu May 23 17:48 - 18:09  (00:20)
reboot   system boot  3.10.0-957.12.2. Thu May 23 17:40 - 18:09  (00:28)

wtmp begins Thu May 23 17:40:54 2019
```

The last command lists all of the logins that have happened on the system since the last time that the /var/log/wtmp file was rotated. It includes where they logged in from, when they logged in and out, and how long they were logged in.

These two commands are useful for seeing who is currently using your system and what users' historical usage looks like. w is especially useful when you want to reboot a shared system - it shows who is currently logged in, giving you the ability to warn them before they get kicked off.

mount

The mount command is used to attach filesystems to a system, but when it is run without any arguments it will list all of the filesystems that are currently mounted:

```
[root@te-master ~]# mount
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
devtmpfs on /dev type devtmpfs
 ↪  (rw,nosuid,size=10268152k,nr_inodes=2567038,mode=755)
securityfs on /sys/kernel/security type securityfs
 ↪  (rw,nosuid,nodev,noexec,relatime)
...
nfsd on /proc/fs/nfsd type nfsd (rw,relatime)
sunrpc on /var/lib/nfs/rpc_pipefs type rpc_pipefs (rw,relatime)
/dev/sda2 on / type xfs (rw,relatime,attr2,inode64,noquota)
172.16.1.253:/data on /data type nfs4 (rw,relatime,vers=4.1,rsize=1048576,w⌋
 ↪  size=1048576,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,clie⌋
 ↪  ntaddr=172.16.1.254,local_lock=none,addr=172.16.1.253)
```

```
tmpfs on /run/user/0 type tmpfs
 ↪  (rw,nosuid,nodev,relatime,size=2055828k,mode=700)
binfmt_misc on /proc/sys/fs/binfmt_misc type binfmt_misc (rw,relatime)
```

Linux systems use the filesystem abstraction for a lot more than just standard storage, so the output of mount may be many lines long. In this example, the two lines that are most immediately interesting are the one that starts with /dev/sda2 and the one that starts with 172.16.1.253. The first one is from the second partition on an actual physical disk named /dev/sdb. This partition is mounted at the root of the filesystem tree (/) and is formatted as an XFS filesystem. The second one is a filesystem that is mounted over the network from 172.16.1.253 and is mounted on the /data directory. We don't know the actual format of this filesystem, as it resides on a different system, but we know that it is using the NFS protocol to do the remote mount.

The mount command can do a lot more than just list mounts, but running it without any arguments makes it a useful introspection tool for finding what filesystems exist on the system, which ones are local or remote, and what the actual hardware is behind each one.

top ("top processes")

The top command is an interactive command that lists all of the processes running on a system, sorting them by by how active they are. By default, it will update once every three or so seconds.

```
[root@te-master ~]# top
top - 22:42:10 up  8:39,  1 user,  load average: 0.00, 0.01, 0.05
Tasks: 235 total,   1 running, 234 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni,100.0 id,  0.0 wa,  0.0 hi,  0.0 si,
 ↪  0.0 st
KiB Mem : 20558264 total, 19299788 free,   612800 used,   645676 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 19502700 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1692 root      20   0       0      0      0 S   0.3  0.0   0:49.82
 ↪  kworker/3:1
10617 root      20   0  162124   2332   1540 R   0.3  0.0   0:00.08 top
    1 root      20   0  192056   4940   2508 S   0.0  0.0   0:03.74 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.00 kthreadd
    3 root      20   0       0      0      0 S   0.0  0.0   0:00.00
 ↪  ksoftirqd/0
    4 root      20   0       0      0      0 S   0.0  0.0   0:00.00
 ↪  kworker/0:0
    5 root       0 -20       0      0      0 S   0.0  0.0   0:00.00
 ↪  kworker/0:+
    7 root      rt   0       0      0      0 S   0.0  0.0   0:00.00
 ↪  migration/0
```

```
    8 root      20   0       0      0      0 S   0.0  0.0   0:00.00 rcu_bh
    9 root      20   0       0      0      0 S   0.0  0.0   0:01.13
↪  rcu_sched
   10 root       0 -20       0      0      0 S   0.0  0.0   0:00.00
↪  lru-add-dr+
   11 root      rt   0       0      0      0 S   0.0  0.0   0:00.14
↪  watchdog/0
```

The first several lines give an overview of the system: the same output as uptime, the number of and state of processes on the system, and some statistics on CPU and memory usage. The bottom part of the screen lists individual processes, which by default are sorted by the %CPU column. The second line in this section is the top command itself, which has process ID 10617 and is being run by root. It has a priority of 20, which is a numeric score used to help the kernel scheduler decide which process to run next. The NI column displays this process's "nice" value, which is another value used to help the kernel schedule processes. This process is currently using a total of 162124 bytes of virtual memory on the system, of which 2332 bytes are in physical memory and 1540 bytes are shared with other processes. This process is currently running, as shown by the R value in the state (S) column, and it is currently using 0.3% of the system's available CPU time and 0.0% of its available memory. It has used 0.08 seconds of CPU time since it started.

The top command is a very powerful tool for seeing what processes are using the most resources on a system and how that resource usage is changing over time. It is a useful command to leave running in a terminal for an "at a glance" view of the system, and it is also useful for quickly seeing why a system is feels bogged down or if a particular process is making progress.

Part 2: Processes

Processes are one of the base units of execution that exist on a Unix-like system. When you run a program, it starts a new *process* that is assigned a unique *process ID* (PID) number and is named after the name of the program being run. Processes use memory and CPU time, and many of them can be running at once.

ps

We have already seen top and how it provides an interactive display of all of the processes running on a system. The ps command can be used to get more information about processes on the command line. ps has a *very* large number of command line options, and we will explore several of them here.

```
[root@te-master ~]# ps
  PID TTY          TIME CMD
11421 pts/0    00:00:00 bash
11461 pts/0    00:00:00 ps
```

When run with no arguments, ps gives a list of processes that are associated with your current login session. In this case, there are two: bash (the login shell) and ps (which is currently running). Their proccess IDs are 11421 and 11461, respectively, and neither have used enough CPU time to even register in the TIME column.

ps -e

To see all of the processes running on the system, you can give ps the −e argument to show *every* process. This output will be noticeably longer than the previous output:

```
[root@te-master ~]# ps -e
  PID TTY          TIME CMD
    1 ?        00:00:12 systemd
    2 ?        00:00:00 kthreadd
    3 ?        00:00:00 ksoftirqd/0
    4 ?        00:00:00 kworker/0:0
    5 ?        00:00:00 kworker/0:0H
...
11407 ?        00:00:00 kworker/7:2
11418 ?        00:00:00 sshd
11421 pts/0    00:00:00 bash
11784 pts/0    00:00:00 ps
21060 ?        00:00:01 kworker/u32:2
```

This gives a better view of the system, but doesn't give an indication of who is running each process.

ps -eF

To get even more information in the ps output, you can add the −F option to enable *full* output lines.

```
[root@te-master ~]# ps -eF
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
root         1     0  0 48014  4940   3 May30 ?        00:00:12
 ↪  /usr/lib/systemd/systemd --switched-root --system --deserialize 22
root         2     0  0     0     0   8 May30 ?        00:00:00 [kthreadd]
root         3     2  0     0     0   0 May30 ?        00:00:00
 ↪  [ksoftirqd/0]
root         4     2  0     0     0   0 May30 ?        00:00:00
 ↪  [kworker/0:0]
root         5     2  0     0     0   0 May30 ?        00:00:00
 ↪  [kworker/0:0H]
...
```

```
root     11407     2  0     0     0   7 09:09 ?        00:00:00
 ↪  [kworker/7:2]
root     11418  4311  0 37118  5448  13 09:10 ?        00:00:00 sshd:
 ↪  root@pts/0
root     11421 11418  0 28893  2092  10 09:10 pts/0    00:00:00 -bash
root     12037 11421  0 38843  1844  12 09:19 pts/0    00:00:00 ps -eF
root     21060     2  0     0     0   4 02:00 ?        00:00:01
 ↪  [kworker/u32:2]
```

This output provides a lot more information about the processes, including the process owner (UID), the process's parent process (PPID), more information about resource utilization, and the complete command line that was used to start the process.

ps -elF

Adding the −l argument adds even more information to the line.

```
[root@te-master ~]# ps -elF
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN    RSS PSR STIME TTY
 ↪      TIME CMD
4 S root         1     0  0  80   0 - 48014 ep_pol  4940   3 May30 ?
 ↪  00:00:12 /usr/lib/systemd/systemd --switched-root --system
 ↪  --deserialize 22
1 S root         2     0  0  80   0 -     0 kthrea     0   8 May30 ?
 ↪  00:00:00 [kthreadd]
1 S root         3     2  0  80   0 -     0 smpboo     0   0 May30 ?
 ↪  00:00:00 [ksoftirqd/0]
1 S root         4     2  0  80   0 -     0 worker     0   0 May30 ?
 ↪  00:00:00 [kworker/0:0]
1 S root         5     2  0  60 -20 -     0 worker     0   0 May30 ?
 ↪  00:00:00 [kworker/0:0H]
...
4 D root     11418  4311  0  80   0 - 37118 flush_  5448  13 09:10 ?
 ↪  00:00:00 sshd: root@pts/0
4 S root     11421 11418  0  80   0 - 28893 do_wai  2092  11 09:10 pts/0
 ↪  00:00:00 -bash
1 S root     12083     2  0  80   0 -     0 worker     0   7 09:20 ?
 ↪  00:00:00 [kworker/7:1]
1 S root     12395     2  0  80   0 -     0 worker     0   7 09:25 ?
 ↪  00:00:00 [kworker/7:2]
4 R root     12940 11421  0  80   0 - 38843 -       1844  14 09:36 pts/0
 ↪  00:00:00 ps -elF
1 S root     21060     2  0  80   0 -     0 worker     0   4 02:00 ?
 ↪  00:00:01 [kworker/u32:2]
```

This extra information includes process state (S), priority (PRI) and nice values (N), and the memory size of the process (SZ).

ps -elF –forest

All processes have a parent process (except for the init process, which is started directly by the kernel). This means that processes can be seen as a tree, or a forest of subtrees rooted on individual processes, when you add the −−forest argument.

```
[root@te-master ~]# ps -elF --forest
...
4 S root      4311     1  0  80   0 - 28216 poll_s  4252   1 May30 ?
 ↪  00:00:00 /usr/sbin/sshd -D
4 S root     11418  4311  0  80   0 - 37118 flush_  5448  13 09:10 ?
 ↪  00:00:00  \_ sshd: root@pts/0
4 S root     11421 11418  0  80   0 - 28893 do_wai  2092  11 09:10 pts/0
 ↪  00:00:00      \_ -bash
0 R root     12988 11421  0  80   0 - 38919 -       2168  14 09:37 pts/0
 ↪  00:00:00          \_ ps -elF --forest
...
```

In this example, all of the initial columns are the same as with −elF, but the last column includes some ASCII art the indicates the lineage of the processes.

ps All the Things!

The ps command is extremely useful and has a lot more options than we have explored here. And the options we looked at here can all be used in different combinations to get different levels of detail. You can read more about the other options available by reading the ps manual page: man ps.

kill

The kill command is used to send signals to processes. Its name comes from its default behavior: to send a signal to a process that will make it terminate.

Trying out kill requires opening a second login session so you have two shells running. In one shell, start a sleep command that will do nothing for 300 seconds:

```
[root@te-master ~]# sleep 300
```

In a second shell, find that process's PID and use kill to terminate it:

```
[root@te-master ~]# ps -e | grep sleep
13801 pts/2    00:00:00 sleep
[root@te-master ~]# kill 13801
```

In your original shell, you should see that the process has been killed:

```
[root@te-master ~]# sleep 300
Terminated
[root@te-master ~]#
```

The kill command can be used to terminate processes that aren't running in your login session (so you can't hit ctrl−C to kill them). It can also be used to harass your teammates by killing their login shells when they least expect it. But that's not nice, so you definitely shouldn't try it at any point. Even if it would be funny.

Part 3: Special filesystems

As we saw in the mount example earlier, Linux (and all Unix-like operating systems) make heavy use of the *file* and *filesystem* abstractions to implement features. Linux has a few special directories that look like standard filesystems, but are actually links into the kernel's memory and configuration. These can be used to get information about data structures within the kernel, and they can be used to change the configuration of a running kernel.

/proc

At its base, the /proc filesystem provides information about every process that is running on the system:

```
[root@te-master ~]# ls /proc
1/      163/   1933/  32/    4310/  4707/  67/    95/          kpageflags
10/     1692/  1940/  33/    4311/  4709/  68/    956/         loadavg
100/    1693/  1961/  3360/  4312/  4710/  69/    96/          locks
102/    1694/  1962/  34/    4319/  4753/  7/     9613/        mdstat
103/    17/    1963/  35/    4325/  4768/  70/    97/          meminfo
104/    170/   1969/  36/    4333/  48/    71/    98/          misc
...
```

Inside the /proc directory is one directory for every process ID on the system, plus some regular files that include information about internal kernel data. Keep in mind that /proc is not a real filesystem, and it does not take up space on disk. Instead, each time you run ls on the directory or look at a file inside it, the kernel intercepts the underlying system calls and returns the information that corresponds with the command you ran.

The /proc filesystem is the standard place to find information about all processes running on a system, and it is the place that the ps command gets its information from.

/proc/<pid>

Each process has a virtual dirctory in /proc, and those directories contain virtual files that hold information about that process. You can use the special shell variable $$ to find the process ID of your login shell and then look at its information:

```
[root@te-master ~]# echo $$
11421
[root@te-master ~]# ls /proc/11421
attr             cwd       map_files   oom_adj        schedstat  task
autogroup        environ   maps        oom_score      sessionid  timers
auxv             exe       mem         oom_score_adj  setgroups  uid_map
cgroup           fd        mountinfo   pagemap        smaps      wchan
clear_refs       fdinfo    mounts      patch_state    stack
cmdline          gid_map   mountstats  personality    stat
comm             io        net         projid_map     statm
coredump_filter  limits    ns          root           status
cpuset           loginuid  numa_maps   sched          syscall
```

You can look at these individual files to get more information about a process. For example, the cmdline file tells you what the process's original command line was

```
[root@te-master ~]# cat /proc/11421/cmdline
-bash
```

Some of these files are special. For example, exe appears as a symbolic link to the binary that the process is running:

```
[root@te-master ~]# ls -l /proc/11421/exe
lrwxrwxrwx 1 root root 0 May 31 10:10 /proc/11421/exe -> /usr/bin/bash
```

Some of these files are *really* special. For example, mem is a file representing the actual memory being used in the kernel by the process.

Tools like ps and top use the information in /proc to display an overview of the processes on a system, but there is a lot of other information in the directory that can be useful for looking more closely at what a process is doing.

/proc/cpuinfo

The /proc filesystem contains a variety of other files that provide insight into the kernel's view of a system. One of these is /proc/cpuinfo:

```
[root@te-master ~]# cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz
stepping        : 1
microcode       : 0xb000017
cpu MHz         : 1200.219
cache size      : 20480 KB
physical id     : 0
siblings        : 16
core id         : 0
cpu cores       : 8
apicid          : 0
initial apicid  : 0
fpu             : yes
...
```

This file contains information about every processor on a system, and can be very long on multi-core systems. This is a good place to look to see how many processors your system has, who manufactured them, and what their speeds are.

/proc/meminfo

The /proc/meminfo file contains more detailed information about system memory usage than free provides:

```
[root@te-master ~]# cat /proc/meminfo
MemTotal:       20558264 kB
MemFree:        19296476 kB
MemAvailable:   19500660 kB
Buffers:            2068 kB
Cached:           191800 kB
SwapCached:            0 kB
Active:           264256 kB
Inactive:         147752 kB
...
```

This file can be useful for getting a more complete view of a system's memory use, especially when you are looking at system preformance and need to know how much total memory is being used and how the kernel is using it.

/proc/self

There is a special /proc directory called self that points to the /proc/pid directory for the current running process.

```
$ ls -l /proc/self
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 /proc/self -> 23371
```

```
$ cat /proc/self/cmdline
cat /proc/self/cmdline
```

That's right, the command is actually the command that inspects the file, because that's the process touching the file. A little confusing, but very handy.

/proc/self/{mounts,mountinfo}

In one of the previous sections we looked at our mount list with the mount command. This can be dangerous if we have a broken mount. The mount comandd could hang forever rather than giving us the information we want. Instead, we can run:

```
$ cat /proc/self/mounts
rootfs / rootfs rw 0 0
sysfs /sys sysfs rw,seclabel,nosuid,nodev,noexec,relatime 0 0
proc /proc proc rw,nosuid,nodev,noexec,relatime 0 0
devtmpfs /dev devtmpfs
 ↪  rw,seclabel,nosuid,size=10267100k,nr_inodes=2566775,mode=755 0 0
securityfs /sys/kernel/security securityfs rw,nosuid,nodev,noexec,relatime
 ↪  0 0
tmpfs /dev/shm tmpfs rw,seclabel,nosuid,nodev 0 0
...
```

For even more info, there is:

```
$ cat /proc/self/mountinfo
17 38 0:17 / /sys rw,nosuid,nodev,noexec,relatime shared:6 - sysfs sysfs
 ↪  rw,seclabel
18 38 0:3 / /proc rw,nosuid,nodev,noexec,relatime shared:5 - proc proc rw
19 38 0:5 / /dev rw,nosuid shared:2 - devtmpfs devtmpfs
 ↪  rw,seclabel,size=10267100k,nr_inodes=2566775,mode=755
20 17 0:16 / /sys/kernel/security rw,nosuid,nodev,noexec,relatime shared:7
 ↪  - securityfs securityfs rw
...
```

/sys

As we noted above, /proc has a virtual directory for each process, but it also has extra virtual files and directories that contain information about the physical system itself. In order to prevent /proc from becoming too cluttered while at the same time making space to add more views into the kernel, a new filesystem was added to contain system-related information: /sys.

```
[root@te-master ~]# ls -F /sys
block/  class/  devices/   fs/          kernel/  power/
bus/    dev/    firmware/  hypervisor/  module/
```

The /sys filesystem is much more hierarchical than /proc and is focused on providing an interface to drivers and hardware devices within the kernel. Interesting directories include:

- /sys/bus - Communication busses on the system, including PCI, USB, and memory hardware
- /sys/fs - Views of the system as seen by filesystem drivers
- /sys/devices/system/cpu - Kernel views of each of the processors on the system

The files in /sys are used by various system utilities as an interface to talking to the kernel and can be used to set or read driver parameters after boot time.

To get an idea of what information you can get from /sys, try:

```
$ ls -l /sys/class/net
total 0
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 em1 ->
 ↪  ../../devices/pci0000:00/0000:00:04.0/net/em1
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 em2 ->
 ↪  ../../devices/pci0000:00/0000:00:03.0/net/em2
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 ens5 ->
 ↪  ../../devices/pci0000:00/0000:00:05.0/net/ens5
lrwxrwxrwx. 1 root root 0 Jun  4 13:41 lo -> ../../devices/virtual/net/lo
```

This gives a listing of the network devices on the system. You can see that the device names are symbolic links to PCI devices by PCI device ID. Let's take a look inside one:

```
$ ls -l /sys/class/net/em1/
total 0
-r--r--r--. 1 root root 4096 Jun  4 13:41 addr_assign_type
-r--r--r--. 1 root root 4096 Jun  4 13:41 address
-r--r--r--. 1 root root 4096 Jun  5 02:58 addr_len
-r--r--r--. 1 root root 4096 Jun  5 02:58 broadcast
-rw-r--r--. 1 root root 4096 Jun  5 02:58 carrier
-r--r--r--. 1 root root 4096 Jun  5 02:58 carrier_changes
lrwxrwxrwx. 1 root root    0 Jun  4 13:41 device -> ../../../0000:00:04.0
... (lots of files omitted)
```

There is a lot of info about the device here, and some of the special files allow you to control the device by, e.g. writing to them. That's take a look inside address:

```
$ cat /sys/class/net/em2/address
de:ad:be:ef:ff:ff
```

This is the MAC address of our ethernet device.

Similar structures exist for many devices and subsystems in the kernel. This can be a great way to inspect deeper system properties, and even write programs or scripts to view and manipulate them.

/dev

The “everything is a file” abstraction model extends to directly accessing physical devices, and the files representing devices themselves (as opposed to the kernel’s driver files in /sys) can be found in /dev.

```
[root@te-master ~]# ls -F /dev
autofs           log=                shm/      tty25  tty48  uhid
block/           loop-control        snapshot  tty26  tty49  uinput
bsg/             mapper/             snd/      tty27  tty5   urandom
btrfs-control    mcelog              stderr@   tty28  tty50  usbmon0
bus/             mem                 stdin@    tty29  tty51  usbmon1
char/            mqueue/             stdout@   tty3   tty52  vcs
console          net/                tty       tty30  tty53  vcs1
core@            network_latency     tty0      tty31  tty54  vcs2
cpu/             network_throughput  tty1      tty32  tty55  vcs3
cpu_dma_latency  null                tty10     tty33  tty56  vcs4
crash            nvram               tty11     tty34  tty57  vcs5
disk/            oldmem              tty12     tty35  tty58  vcs6
dri/             port                tty13     tty36  tty59  vcsa
fb0              ppp                 tty14     tty37  tty6   vcsa1
fd@              ptmx                tty15     tty38  tty60  vcsa2
full             ptp0                tty16     tty39  tty61  vcsa3
fuse             pts/                tty17     tty4   tty62  vcsa4
hpet             random              tty18     tty40  tty63  vcsa5
hugepages/       raw/                tty19     tty41  tty7   vcsa6
hwrng            rtc@                tty2      tty42  tty8   vfio/
infiniband/      rtc0                tty20     tty43  tty9   vga_arbiter
initctl@         sda                 tty21     tty44  ttyS0  vhci
input/           sda1                tty22     tty45  ttyS1  vhost-net
kmsg             sda2                tty23     tty46  ttyS2  zero
knem             sg0                 tty24     tty47  ttyS3
```

The files in /dev are connected to their physical devices at a very low level, so you must take care when working with them. For example, /dev/sda is the actual boot drive on this system, and copying another file on top of it would most likely make the root filesystem unusable.

A lot of device functions are controlled by making special ioctl (input-output control) syscalls on the files in /dev. ioctl is a file syscall like write. It allows an interface like syscall, but for specific device functions.

After looking at these three filesystems, it has probably become clear that there is a lot of overlap between the files they contain. This is due to the iterative way in which they were developed - as /proc and /dev got cluttered with files that didn’t have to do with processes or devices, /sys was developed to help the situation. But backward compatibility is still important, so legacy files exist in both of the other locations that may not make sense there anymore. Writing perfect software is hard.

Part 4: Logging

Linux systems are dynamic systems that are constantly changing, but most of the tools and files we have looked at so far only look at a single point in time. Systems provide a few logging tools that are useful for finding information about events that have already passed.

dmesg

The dmesg command prints out the kernel's log buffer, which is frequently very long. This log starts at boot time and, until it gets too long and is truncated, goes up to the present time.

```
[root@te-master ~]# dmesg
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.10.0-957.12.2.el7.x86_64
 ↪  (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red
 ↪  Hat 4.8.5-36) (GCC) ) #1 SMP Tue May 14 21:24:32 UTC 2019
[    0.000000] Command line:
 ↪  BOOT_IMAGE=/boot/vmlinuz-3.10.0-957.12.2.el7.x86_64
 ↪  root=UUID=c0c1de66-90c5-4515-8410-7cc166030bd6 ro
[    0.000000] e820: BIOS-provided physical RAM map:
...
[    6.853739] IPv6: ADDRCONF(NETDEV_UP): ib0: link is not ready
[    6.860594] IPv6: ADDRCONF(NETDEV_CHANGE): ib0: link becomes ready
[    7.451210] e1000: em2 NIC Link is Up 1000 Mbps Full Duplex, Flow
 ↪  Control: RX
[    7.456375] IPv6: ADDRCONF(NETDEV_CHANGE): em2: link becomes ready
[    7.464115] e1000: em1 NIC Link is Up 1000 Mbps Full Duplex, Flow
 ↪  Control: RX
[    7.477089] e1000: ens5 NIC Link is Up 1000 Mbps Full Duplex, Flow
 ↪  Control: RX
[    7.481344] IPv6: ADDRCONF(NETDEV_CHANGE): em1: link becomes ready
[    7.486775] IPv6: ADDRCONF(NETDEV_CHANGE): ens5: link becomes ready
```

The first column of dmesg output is a timestamp measured in seconds since the kernel started booting. The rest of each line is free-form text that was provided by the kernel subcomponent that printed it.

The dmesg command is a good place to start when you suspect something is wrong with a system at the kernel level. It tends to be fairly quiet after boot time, but the kernel will use this buffer to log any problems or unusual circumstances it sees.

There are a number of ways you can view the dmesg output with different options. One particularly handy one is −H:

```
[Jun 4 13:41] Initializing cgroup subsys cpuset
[  +0.000000] Initializing cgroup subsys cpu
[  +0.000000] Initializing cgroup subsys cpuacct
[  +0.000000] Linux version 3.10.0-957.12.2.el7.x86_64
 ↪  (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red
 ↪  Hat 4.8.5-36)
[  +0.000000] Command line:
 ↪  BOOT_IMAGE=/boot/vmlinuz-3.10.0-957.12.2.el7.x86_64
 ↪  root=UUID=c0c1de66-90c5-4515-8410-7cc166030bd6 ro
[  +0.000000] e820: BIOS-provided physical RAM map:
[  +0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable
[  +0.000000] BIOS-e820: [mem 0x000000000009fc00-0x000000000009ffff]
 ↪  reserved
[  +0.000000] BIOS-e820: [mem 0x00000000000f0000-0x00000000000fffff]
 ↪  reserved
```

This gives a more human-readable output, with a pager and color (which you can't see in this guide).

journalctl

While the dmesg command prints out the contents of the kernel's log buffer, the journalctl command is used to examine the general-purpose logging infrastructure provided by systemd. These logs are stored in a special binary format that makes them easy to search through and filter, but makes it impossible to use traditional Unix tools like grep, sed, and awk on them directly.

```
[root@te-master ~]# journalctl
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
 ↪  21:31:48 MDT. --
May 30 14:02:19 te-master systemd-journal[162]: Runtime journal is using
 ↪  8.0M (max allowed 1003.8M, trying to leave 1.4G free of 9.7G available
 ↪  → current limit 1003.8M).
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpuset
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpu
May 30 14:02:19 te-master kernel: Initializing cgroup subsys cpuacct
May 30 14:02:19 te-master kernel: Linux version 3.10.0-957.12.2.el7.x86_64
 ↪  (mockbuild@kbuilder.bsys.centos.org) (gcc version 4.8.5 20150623 (Red
 ↪  Hat 4.8.5-36) (GCC) ) #1 SMP Tue May 14 21:24:32 UTC 2019
...
```

When run with no arguments, journalctl will display all of the logs that it knows about. Each line starts with a timestamp, followed by the name of the system the log entry was generated on (te−master) and the component that generated it (systemd−journal and kernel in this example). The rest of the line is then a free-form set of text that was sent by the component as its log message.

Note that, by default, journalctl will send its output through a pager (less, usually), so you will need to hit the q key to exit the display after running this command.

One handy option is to add −r to get log entries in reverse order (i.e. most recent on top).

journalctl -f

By adding the −f option to journalctl, you can *follow* the journal as it is written to. The output will start with a number of recently written lines, at which point the command will pause and wait for more log lines to be written to the journal. It will then print out new lines in real time as they come in.

Since it is impossible to display dynamic text in this guide, you should try this out by following these steps:

1. In one terminal window, run journalctl −f on your master node. Let it continue running.
2. In a second terminal window, log in to your master node a second time.
3. You should see log messages generated by your second login appear on your journalctl −f output.

journalctl -u

By default, journalctl will display the output of all systemd units that wrote to it. You can filter it to only show entries from specific units by using the −u option.

```
[root@te-master ~]# journalctl -u sshd
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
 ↪  21:39:29 MDT. --
May 30 14:02:25 te-master systemd[1]: Starting OpenSSH server daemon...
May 30 14:02:25 te-master sshd[4311]: Server listening on 0.0.0.0 port 22.
May 30 14:02:25 te-master sshd[4311]: Server listening on :: port 22.
May 30 14:02:25 te-master systemd[1]: Started OpenSSH server daemon.
May 30 14:03:49 te-master sshd[5063]: Accepted publickey for root from
 ↪  172.16.1.253 port 48810
May 30 14:44:25 te-master sshd[5301]: Accepted publickey for root from
 ↪  172.16.1.253 port 48812
May 30 14:46:18 te-master sshd[5420]: Accepted publickey for root from
 ↪  172.16.1.253 port 48814
```

In this example, we are looking only at the output related to the sshd unit. Filtering by unit is very useful when you are trying to figure out why a service doesn't appear to be working correctly, or when you want to see what activity has been happening with a service like ssh.

journalctl –since and –until

The journalctl command can filter in time ranges using the −−since and −−until options. These can be used individually, or they can be used together to filter within a very specific window of time.

```
[root@te-master ~]# journalctl --since "2019-05-31 00:00:00" --until
 ↪  "2019-05-31 12:00:00"
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
 ↪  21:46:29 MDT. --
May 31 00:01:01 te-master systemd[1]: Started Session 14 of user root.
May 31 00:01:01 te-master CROND[14801]: (root) CMD (run-parts
 ↪  /etc/cron.hourly)
May 31 00:01:01 te-master run-parts(/etc/cron.hourly)[14804]: starting
 ↪  0anacron
May 31 00:01:01 te-master anacron[14810]: Anacron started on 2019-05-31
May 31 00:01:01 te-master anacron[14810]: Normal exit (0 jobs run)
May 31 00:01:01 te-master run-parts(/etc/cron.hourly)[14812]: finished
 ↪  0anacron
May 31 00:03:27 te-master munged[4917]: Purged 10 credentials from replay
 ↪  hash
May 31 00:06:27 te-master munged[4917]: Purged 20 credentials from replay
 ↪  hash
May 31 00:09:27 te-master munged[4917]: Purged 10 credentials from replay
 ↪  hash
...
```

In this example, we are just looking at log messages that came in between midnight and noon on May 31. If we had left out −−since, it would have started at the earliest messages available. If we had left out −−until, it would have ended at the current time.

The journalctl command has many, many more ways to filter. You can find more in its man page (man journalctl). Note that you can use many commands together to create complex filters. For example, we can use −u, −−since, and −−until together to find all ssh activity between midnight and noon on May 31:

```
[root@te-master ~]# journalctl -u sshd --since "2019-05-31 00:00:00"
 ↪  --until "2019-05-31 12:00:00"
-- Logs begin at Thu 2019-05-30 14:02:19 MDT, end at Sat 2019-06-01
 ↪  21:49:29 MDT. --
May 31 09:10:08 te-master sshd[11418]: Accepted publickey for root from
 ↪  172.16.1.253 port 48818
May 31 09:51:45 te-master sshd[13752]: Accepted publickey for root from
 ↪  172.16.1.253 port 48820
```

/var/log

While the binary format used by journald makes it conventient to filter and search logs, it also makes it difficult to send the logs through other programs for analysis. The /var/log directory contains plain text copies of system log files, frequently broken out by the component that generated them:

```
audit/             dmesg              mariadb/           tallylog
boot.log           dmesg.old          messages           tuned/
boot.log-20190531  firewalld          munge/             wtmp
btmp               grubby_prune_debug ntpstats/          yum.log
conman/            httpd/             secure             yum.log-20190530
conman.log         lastlog            slurm_jobacct.log
cron               maillog            spooler
```

These logs are generated by the syslog service, and which logs are written into which files is controlled by its configuration. The overall contents of these files are substantially similar to those found with journalctl, so deciding which place to look for information is generally based on how you plan to process the data.

Part 5: System Control

- reboot, halt, systemctl reboot, systemctl poweroff

All good things must come to an end, and you will eventually need to either reboot or shut down one of your systems. These actions can both be done two different ways: using systemctl, and using traditional Unix commands. There is no particular advantage to either option - choose what you like the best.

The traditional way

```
[root@te-master ~]# reboot
```

The reboot command cleanly restarts your system. If you are logged in over via ssh, your connection will immediately drop and you won't be able to log in again until the system has restarted. Make sure your network is configured correctly before doing this so you will be able to log in again!

```
[root@te-master ~]# halt -p
```

The halt command shuts down the system, but does not restart it. When the −p option is included, it also powers the system off. After running the halt command, you will no longer have access to your system until you physically power it back on, either via remote power commands or by pushing the button on its front panel.

The systemd way

```
[root@te-master ~]# systemctl reboot
```

This performs the same action as reboot, but uses systemd syntax to do it.

```
[root@te-master ~]# systemctl poweroff
```

This performs the same action as halt −p, but uses systemd syntax to do it.

Networks and Services

Overview

This guide will walk you through two related activities:

1. Configure the cluster network between your master node and your compute node. This is the network that will be used for managing your system.
2. Work with some of the base services that make your cluster work: ssh, ntp, and http.
3. Finally, we'll take a look at how some practical systemd internals work.

Assumptions

This guide assumes that the following steps have already been completed:

- The master node is built and has external network connectivity on interface em2.
- One compute node is built with the address 172.16.0.1 on interface em1.
- The compute node has its default route going through the master's cluster address: 172.16.0.254.

Step 1: Examine your network configuration

We will start by looking at your current network configuration and some useful tools for testing that it is working correctly.

1. Look at your master's network configuration. On the master node, run:

   ```
   ip addr show
   ```

   This will show you information about each network interface on your system. We'll look at some of its output in this tutorial, but you will need to explore and use other parts of its output on your own.

2. Look for the line that starts with em2. This section describes your external interface, and it should have an address that starts with 10.0.52. This interface is already configured and does not need to be changed.

3. Check your nodes' routing table. What is your default route?

   ```
   # ip route
   default via 10.0.52.1 dev em2 proto static metric 101
   10.0.52.0/24 dev em2 proto kernel scope link src 10.0.52.147 metric 101
   ```

   We see our default route goes through 10.0.52.1 on em2. The route for the network 10.0.52.0/24 tells us how to get to the 10.0.52.1 address.

4. Test your master node's external connectivity. First, try looking up Google's public nameserver:

   ```
   nslookup 8.8.8.8
   ```

   This should return results similar to:

   ```
   Server:    10.0.35.34
   Address:   10.0.35.34#53
   Non-authoritative answer:
   8.8.8.8.in-addr.arpa name = google-public-dns-a.google.com.
   ```

   If this fails, install the bind−utils package on your master node and try again:

   ```
   yum install -y bind-utils
   ```

   This test ensures that you are able to contact your own nameserver and look up the name of an external system.

5. Next, trace the route from your master to that same public system. For this we need the traceroute package:

   ```
   yum install -y traceroute
   ```

   Now, traceroute to 8.8.8.8:

   ```
   traceroute 8.8.8.8
   ```

   The output should look similar to this, but will likely be somewhat different:

   ```
   traceroute to 8.8.8.8 (8.8.8.8), 30 hops max, 60 byte packets
    1  usrcmain-switch.usrc (10.15.1.254)  0.236 ms  0.188 ms  0.134 ms
    2  10.15.0.1 (10.15.0.1)  0.674 ms  0.665 ms  0.601 ms
    3  10.254.253.1 (10.254.253.1)  3.074 ms  3.078 ms  3.026 ms
    4  nxgateway-nmconsortium.lanl.gov (192.65.95.133)  1.917 ms  1.929
   ↪  ms  1.962 ms
    5  esnet-g-abq.lanl.gov (192.65.95.2)  2.745 ms  2.964 ms  3.211 ms
    6  denvcr5-ip-a-albqcr5.es.net (134.55.43.114)  10.990 ms  11.021 ms
   ↪  11.152 ms
   ```

```
 7  pnwgcr5-ip-c-denvcr5.es.net (134.55.42.37)  35.978 ms  36.379 ms
 ↪  36.049 ms
 8  google--pnwg-cr5.es.net (198.129.77.201)  54.532 ms  54.506 ms
 ↪  54.497 ms
 9  * * *
10  google-public-dns-a.google.com (8.8.8.8)  54.502 ms  54.485 ms
 ↪  54.508 ms
```

The traceroute command exploits the IP protocol's Time To Live (TTL) field to find the hops between your system and another system. When a switch gets a packet with an expired TTL, it responds saying that it is dropping the packet. By starting with a TTL of 1 and watching for these responses, the traceroute command can figure out the whole path by incrementing the TTL by one with each iteration.

These tests should show that your master has external connectivity and is ready to have its internal cluster network configured.

Step 2: Configure the cluster network on your master node

The cluster network a general-purpose 1Gb network used for cluster management, monitoring, logging, and any other task that doesn't require the use of the high-speed Infiniband network. To start with, you will use it to access your compute node from your master node.

1. List the interfaces on your master node again:

   ```
   ip addr show
   ```

   Look for a line that starts with em1. This will be your cluster network interface and is the interface that we will be configuring right now.

2. Configure em1 on your master node by editing /etc/sysconfig/network−scripts/ifcfg−em1. Its contents should be (the order of lines doesn't matter):

   ```
   TYPE=Ethernet
   PROXY_METHOD=none
   BROWSER_ONLY=no
   BOOTPROTO=static
   DEFROUTE=no
   IPV4_FAILURE_FATAL=no
   IPV6INIT=no
   NAME=em1
   DEVICE=em1
   ONBOOT=yes
   IPADDR=172.16.0.254
   NETMASK=255.255.255.0
   ```

3. Bring down interface em1 and ensure it has no configuration:

```
ifdown em1
ip addr show em1
```

4. Bring interface em1 up and ensure it has the cluster network configuration:

   ```
   ifup em1
   ip addr show em1
   ```

   When configured, your interface should look similar to the following:

   ```
   3: em1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast
    ↪  state UP group default qlen 1000
       link/ether de:ad:be:ef:00:00 brd ff:ff:ff:ff:ff:ff
       inet 172.16.0.254/24 brd 172.16.0.255 scope global noprefixroute
    ↪  em1
          valid_lft forever preferred_lft forever
       inet6 fe80::dcad:beff:feef:0/64 scope link
          valid_lft forever preferred_lft forever
   ```

5. Use ethtool to find more information about your network interface:

   ```
   ethtool em1
   ```

   This will show lots of information about the interface. Some useful things to check are the Speed, Duplex, and Link detected fields to make sure they are 1000Mb/s, Full, and yes, respectively.

   Once your master node's cluster network is configured, you should have connectivity to your compute node. You can test this with the ping command, which sends small packets to the system and watches for responses (use ctrl −C to stop pinging):

   ```
   ping 172.16.0.1
   ```

   The expected output is similar to this:

   ```
   PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.
   64 bytes from 172.16.0.1: icmp_seq=1 ttl=64 time=0.028 ms
   64 bytes from 172.16.0.1: icmp_seq=2 ttl=64 time=0.071 ms
   ...
   ```

   If you see similar results, your cluster network is working!

   Bad output may look similar to this:

   ```
   PING 172.16.0.1 (172.16.0.1) 56(84) bytes of data.
   From 172.16.0.254 icmp_seq=1 Destination Host Unreachable
   From 172.16.0.254 icmp_seq=1 Destination Host Unreachable
   ...
   ```

   If you get this or any other non-successful output, something is not set up correctly. Examine the output of ip addr show and ip link show for clues, or flag down the instructor or a TA for help.

Step 3: Configure NAT on your master

Your cluster's compute nodes will live on a private network that, by default, is unable to access the internet as a whole. We will set up Network Address Translation (NAT) on your master node so it can act as a router for the rest of your nodes. We will use iptables to do this, which is a tool for manipulating the kernel's firewall tables.

1. First, you need to enable IPv4 packet forwarding on your master. You can do this in a single command by running:

   ```
   sysctl -w net.ipv4.ip_forward=1
   ```

   But this will not persist across reboots. To do this persistently, create a new file at /etc/sysctl.d/90−nat.conf:

   ```
   vi /etc/sysctl.d/90-nat.conf
   ```

   Paste the following contents into this file:

   ```
   # Enable IP forwarding for NAT
   net.ipv4.ip_forward = 1
   ```

   This file will get read at boot time to set this parameter correctly, but you can force a reload by running:

   ```
   sysctl --system
   ```

   You can then check that the correct value is set by running:

   ```
   sysctl -n net.ipv4.ip_forward
   ```

   If you get back a single line that says 1, then you are all set! If you get back 0, you should check your work for typos and try again.

   Next, we will install and enable iptables itself. Before we do any work, you can check your existing iptables configuration and confirm that it is empty.

2. Install the iptables packages on your master:

   ```
   yum install -y iptables iptables-services
   ```

3. On your master, run:

   ```
   iptables -L -v
   ```

   The output should look similar to:

   ```
   Chain INPUT (policy ACCEPT 181 packets, 27513 bytes)
   pkts bytes target     prot opt in     out     source
    ↪  destination
   Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
    pkts bytes target     prot opt in     out     source
    ↪  destination
   ```

```
Chain OUTPUT (policy ACCEPT 166 packets, 24553 bytes)
 pkts bytes target     prot opt in     out     source
 ↪  destination
```

Once we have enabled iptables, we'll look at the output again and see there there is more to it.

4. On your master, open /etc/sysconfig/iptables for editing:

   ```
   vi /etc/sysconfig/iptables
   ```

   The file will contain an initial iptables configuration that is more restrictive than we need. Delete all of the lines in the file (you can do this in vi by typing dd repeatedly, or by typing dG while on the first line), and replace them with:

   ```
   *filter
   :INPUT ACCEPT [0:0]
   :FORWARD ACCEPT [0:0]
   :OUTPUT ACCEPT [0:0]
   # Enable forwarding for NAT
   -A FORWARD -i em1 -j ACCEPT
   -A FORWARD -o em1 -j ACCEPT
   COMMIT
   *nat
   :PREROUTING ACCEPT [0:0]
   :OUTPUT ACCEPT [0:0]
   :POSTROUTING ACCEPT [0:0]
   # Enable masquerading for NAT
   -A POSTROUTING -o em2 -j MASQUERADE
   COMMIT
   ```

   Save the file when you are done.

5. On your master, enable and start iptables:

   ```
   systemctl enable iptables
   systemctl start iptables
   ```

Now you can run iptables −L −v again to see some of your rules in place:

```
...
Chain FORWARD (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source
 ↪  destination
    2   168 ACCEPT     all  --  em1    any     anywhere             anywhere
    2   168 ACCEPT     all  --  any    em1     anywhere             anywhere
...
```

And run iptables −L −v −t nat to see the final one:

```
...
Chain POSTROUTING (policy ACCEPT 207 packets, 12218 bytes)
 pkts bytes target     prot opt in     out     source
 ↪  destination
    1    84 MASQUERADE  all  --  any    em2     anywhere
 ↪  anywhere
```

The −t nat option lists the entries in the nat “table.” By default, entries of the filter table are displayed. There is also a mangle table by default. The lists of rules in a table are called a chain. The chains are logically collected into separate tables, because tables provide a convenient way to collect rules that need to happen at specific stages of the filtering process. For instance, the nat POSTROUTING chain needs to be the last chain processed before a packet leaves the system.

With those rules in place, your compute nodes will now have access to the internet via your master node. Test this using ping on your compute node:

```
ping 8.8.8.8
```

Step 4: Set up an SSH key for root

Within the boundaries of a cluster, it is convenient for the root user to be able to log in to compute nodes from the master node without needing to input a password. We will set up host-based ssh access on a later date, but for now, we will use public key authentication for the same convenience.

1. On your compute node, ensure inbound public key authentication is enabled. Look for this line in /etc/ssh/sshd_config:

   ```
   PubkeyAuthentication yes
   ```

   Allowing public key authentication is on by default, so if this line is missing or is commented out, it is still enabled. If the line is there and is set to no, then you should change it to yes. If you need to change the line, be sure to restart the server afterward:

   ```
   systemctl restart sshd
   ```

2. On your master node, generate a public/private key pair for your root user. As root, run:

   ```
   ssh-keygen -t rsa -b 4096
   ```

   This will create a key of type RSA and of length 4096 bits. When prompted for the location to store the key, hit enter to accept the default (/root/.ssh/id_rsa). When prompted (twice) to enter a password, hit enter (twice) to use a blank password.

3. Copy the public portion of the key to the compute node, putting it in root’s authorized_keys file. On your compute node:

```
cd $HOME
mkdir .ssh
cd .ssh
vi authorized_keys
```

Paste into that file the contents of /root/.ssh/id_rsa.pub. Make sure it's all on one line! Save the file. Note that this file can contain multiple public keys, each on a separate line. For now, we only need the one key we just pasted in.

It is important that the authorized_keys file only be readable by the user that owns it. SSH will ignore it if it is readable by anyone else. Set:

```
chmod 0600 ~/.ssh/authorized_keys
```

4. To confirm that your public key is working, run ssh in verbose mode on your master:

```
ssh -v 172.16.0.1
```

Near the end of its output, you should see a line that shows your public key authentication succeeding:

```
...
debug1: Authentication succeeded (publickey).
...
```

Step 5: Configure NTP

NTP, the network time protocol, is a convenient service that ensures a set of systems all have identical time. We will set up your master to get its time from an external time server and to serve its time to the cluster nodes.

1. During the install process, your master node might have been automatically set up to synchronize its time from a pool of servers managed by the Centos project. This is a good place to start with the master. You can test it by running, on the master:

```
ntpd -q
```

After running for a few seconds, that command will print out a line showing how much it changed your clock by:

```
ntpd: time slew -0.001373s
```

If NTP was not installed automatically when you built your master node, the above test will fail:

```
[root@localhost ~]# ntpd -q
-bash: ntpd: command not found
```

In this case, install the NTP server and try again. On the master, run:

```
yum install -y ntp
ntpd -q
```

2. After ntpd is installed, ensure it is enabled to start at boot time. On your master node, run:

   ```
   systemctl list-unit-files
   ```

   Scroll down until you find the line for ntpd and ensure it is enabled:

   ```
   ntpd.service                                  enabled
   ```

   If it is not enabled, tell systemd to enable it and then check again:

   ```
   systemctl enable ntpd
   systemctl list-unit-files
   ```

3. Once NTP is running on your master node, confirm that it is installed on your compute node. If not, install it and enable it by running on the compute node:

   ```
   yum install -y ntp
   systemctl enable ntpd
   ```

4. On your compute node, open /etc/ntp.conf:

   ```
   vi /etc/ntp.conf
   ```

5. Remove any existing lines that start with server. In their place, add a new line that points to your master node:

   ```
   server 172.16.0.254 iburst
   ```

6. Still on your compute node, save the file, restart the NTP service, and test it.

   ```
   systemctl restart ntpd
   ntpd -q
   ```

   Like on your master node, this should run for a few seconds and then tell you how much time it added or removed from the clock.

7. You can get even more information about the ntp status with the ntpq command:

```
# ntpq
ntpq> lpeer
     remote           refid      st t when poll reach   delay   offset
 ↪  jitter
===========================================================================⌋
 ↪  ===
 zeit.arpnetwork .XFAC.          16 u    - 1024    0    0.000    0.000
 ↪  0.000
 eterna.binary.n .XFAC.          16 u    - 1024    0    0.000    0.000
 ↪  0.000
```

```
ntp2.wiktel.com .XFAC.          16 u    - 1024    0    0.000    0.000
↪  0.000
rdl1126.your.or .XFAC.          16 u    - 1024    0    0.000    0.000
↪  0.000
```

Note: this example actually shows a system that *can't* contact these four public NTP servers.

Step 6: Install a web server on your master

nginx (pronounced *engine-X*) is a lightweight HTTP server. We won't use it immediately, but it is a useful service that we will configure later in this course.

1. On your master node, install nginx:

   ```
   yum install nginx
   ```

2. Once it is installed, check its status with systemd:

   ```
   systemctl status nginx
   ```

   This should show that it is disabled and not running:

   ```
   Loaded: loaded (/usr/lib/systemd/system/nginx.service; disabled;
    ↪  vendor preset: disabled)
   Active: inactive (dead)
   ```

3. Enable the service, and then start it:

   ```
   systemctl enable nginx
   systemctl start nginx
   ```

4. On your compute node, check that your web server works:

   ```
   curl http://172.16.0.254
   ```

   You should get back the HTML of the initial welcome page.

5. Add your own "hello world" file to your web server's root directory. On your master, run vi /usr/share/nginx/html/hello.html and add the following contents:

   ```
   <html>
     <head>
       <title>Hello!</title>
     </head>
     <body>
       Hello, world!
     </body>
   </html>
   ```

6. On your compute node, you can get your new page by running:

   ```
   curl http://172.16.0.254/hello.html
   ```

7. HTTP, the protocol used by web servers, is a simple protocol that you can interact with manually. We will do that using the telnet program. On your master, install telnet:

   ```
   yum install -y telnet
   ```

   HTTP runs on port 80. You can connect to that port by running the following on your master:

   ```
   telnet 172.16.0.254 80
   ```

   This will leave you at a blank prompt that is connected to port 80 on your master. Here you can type HTTP protocol commands and see the results. For example, try:

   ```
   GET /hello.html
   ```

   You should see the contents of your hello file, and the connection will close. This is what happens inside the curl command we used earlier.

   There's no need to do any more nginx configuration right now, but you can look at its configuration in /etc/nginx/nginx.conf. You can also put more html files in /usr/share/nginx/html.

Step 7: Systemd, under the hood

Now that we have some services configured, let's take a look at some more systemd specifics.

1. What happens when systemd enables a service? We just enabled nginx. Let's see what that did. There is a special directory where enabled units are controlled:

   ```
   # ls -l /etc/systemd/system
   total 4
   drwxr-xr-x. 2 root root   57 May 23 17:16 basic.target.wants
   lrwxrwxrwx. 1 root root   46 May 23 17:15
    ↪  dbus-org.freedesktop.NetworkManager.service ->
    ↪  /usr/lib/systemd/system/NetworkManager.service
   lrwxrwxrwx. 1 root root   57 May 23 17:15
    ↪  dbus-org.freedesktop.nm-dispatcher.service ->
    ↪  /usr/lib/systemd/system/NetworkManager-dispatcher.service
   lrwxrwxrwx. 1 root root   40 May 23 17:15 default.target ->
    ↪  /usr/lib/systemd/system/multi-user.target
   drwxr-xr-x. 2 root root   87 May 23 17:15 default.target.wants
   drwxr-xr-x. 2 root root   32 May 23 17:15 getty.target.wants
   drwxr-xr-x. 2 root root   35 May 23 17:15 local-fs.target.wants
   drwxr-xr-x. 2 root root 4096 May 26 12:06 multi-user.target.wants
   drwxr-xr-x. 2 root root   48 May 23 17:15 network-online.target.wants
   drwxr-xr-x. 2 root root   31 May 26 12:03 remote-fs.target.wants
   drwxr-xr-x. 2 root root   28 May 26 12:03 sockets.target.wants
   ```

```
drwxr-xr-x. 2 root root  134 May 23 17:15 sysinit.target.wants
drwxr-xr-x. 2 root root   44 May 23 17:15 system-update.target.wants
```

This is a list of all of the targets defined on our system. You can think of targets as defined milestones in the boot process. for a server. Let's take a look inside:

```
# ls -l /etc/systemd/system/multi-user.target.wants/
total 0
lrwxrwxrwx. 1 root root 38 May 23 17:15 auditd.service ->
 ↪  /usr/lib/systemd/system/auditd.service
lrwxrwxrwx. 1 root root 37 May 23 17:15 crond.service ->
 ↪  /usr/lib/systemd/system/crond.service
lrwxrwxrwx. 1 root root 42 May 23 17:16 irqbalance.service ->
 ↪  /usr/lib/systemd/system/irqbalance.service
lrwxrwxrwx. 1 root root 37 May 23 17:15 kdump.service ->
 ↪  /usr/lib/systemd/system/kdump.service
lrwxrwxrwx. 1 root root 46 May 23 17:15 NetworkManager.service ->
 ↪  /usr/lib/systemd/system/NetworkManager.service
lrwxrwxrwx. 1 root root 41 May 26 12:03 nfs-client.target ->
 ↪  /usr/lib/systemd/system/nfs-client.target
lrwxrwxrwx. 1 root root 37 May 26 12:06 nginx.service ->
 ↪  /usr/lib/systemd/system/nginx.service
...
```

We see that there's an nginx symbolic link in this directory. *This* is what happened when we enabled the service; systemctl created this symbolic link.

2. This also shows where the nginx unit file lives. First, run:

   ```
   ls /usr/lib/systemd/system
   ```

   This is where almost all unit files go that are installed by system software (i.e. things that are installed by an RPM). There are a *lot* of them.

   Let's take a look at the nginx one:

   ```
   # cat /usr/lib/systemd/system/nginx.service
   [Unit]
   Description=The nginx HTTP and reverse proxy server
   After=network.target remote-fs.target nss-lookup.target

   [Service]
   Type=forking
   PIDFile=/run/nginx.pid
   # Nginx will fail to start if /run/nginx.pid already exists but has
    ↪  the wrong
   # SELinux context. This might happen when running `nginx -t` from the
    ↪  cmdline.
   # https://bugzilla.redhat.com/show_bug.cgi?id=1268621
   ```

```
ExecStartPre=/usr/bin/rm -f /run/nginx.pid
ExecStartPre=/usr/sbin/nginx -t
ExecStart=/usr/sbin/nginx
ExecReload=/bin/kill -s HUP $MAINPID
KillSignal=SIGQUIT
TimeoutStopSec=5
KillMode=process
PrivateTmp=true

[Install]
WantedBy=multi-user.target
```

Notice the default target on our system is multi−user.target. Generally, this is the target you want to reach

3. Let's suppose that we want to insist that nginx start *After* ssh. We don't actually care about this but suppose we did. How could we do it?

   One option would be to modify the unit file: /usr/lib/systemd/system/nginx.service. However, this is not very maintainable, since it was installed by the RPM, and we generally want the package maintainer to decide what the unit file looks like.

   Because of this, there is a mechanism called an "override" we can do with a unit file. We can override the entire unit file like this:

   ```
   # cp -v /usr/lib/systemd/system/nginx.service /etc/systemd/system
   '/usr/lib/systemd/system/nginx.service' ->
    ↪  '/etc/systemd/system/nginx.service'
   ```

   If we now edit /etc/systemd/system/nginx.service, systemd will use that file instead.

   But, we can do something a bit better. We can override just specific things while letting the package maintainer do the rest. To do this:

   ```
   # mkdir /etc/systemd/nginx.service.d
   # cd /etc/systemd/nginx.service.d
   ```

   Then create the file after−sshd.conf with the following contents:

   ```
   [Unit]
   After=sshd.service
   ```

   This will make nginx load after sshd without modifying anything else. Note that this *appends* to the After= list that already exists (we could clear it if we wanted to).

   The point here is that /etc/systemd/system is where admins are expected to make configuration changes to units. /usr/lib/systemd/system is where package maintainers are expected to install unit files for their packages. It's best not to modify /usr/lib/systemd/system.

DONE

At this point, your two nodes should have the following capabilities:

- The master node should have an external and an internal network interface and should be able to access the internet on its external interface
- The compute node should have an internal interface and should be able to access the internet by using the master node as a router
- Both nodes should be synchronizing their time: the master node to the CentOS servers and the compute node to the master node
- You should be able to use a passwordless SSH key to log in from the master node to the compute node
- You should have a working web server available on your master node

Netboot cluster guide

Overview

This guide contains the steps necessary to get the master to manually PXE boot a single stateless node. This guide is meant as a side reference, and the class should follow the presentation slides.

Assumptions

This guide assumes that the following steps have already been completed:

- CentOS 7 has been installed on the master and updated.
- Interface em2 has been configured as the uplink.
- Interface em1 has been statically configured with IP 172.16.0.254/24 and connected to the cluster switch.
- The MAC address for the first compute node's iDRAC has been collected, and the iDRAC login set to admin/admin.

Step 0: Preliminaries

The following steps will get the system prepared for our pxeboot setup.

1. There is supplementary content on the google drive. Copy the file netboot.tar.gz to your master. Extract it in /root:

   ```
   # tar zxvf netboot.tar.gz -C /root
   ```

This will create the directory /root/netboot that we will be using for several steps, and populate it with some files we'll need.

2. Disable selinux:

   ```
   # setenforce 0
   # vi /etc/sysconfig/selinux
   ```

   Change the line:

   SELINUX=enforcing

   to

   SELINUX=disabled

3. Disable firewall (firewalld):

   ```
   # systemctl stop firewalld
   # systemctl disable firewalld
   ```

4. Install some useful utilities:

   ```
   # yum install tmux lsof links
   ```

Step 1: Configuring DHCP

The heart of the PXEBoot process is controlled through DHCP. DHCP not only has the responsibility of handing out IP addresses, but also instructing the node as to where it can find a pxe executable (pxelinux.0) file that will launch the PXE process.

1. Install dhcp:

   ```
   yum install dhcp
   ```

2. Configure dhcp. DHCPd is controlled through the file /etc/dhcp/dhcpd.conf. Open this file and create the following contents (a copy of this file can be found at /root/netboot/master/etc/dhcp/dhcpd.conf):

   ```
   default-lease-time 7200;
   max-lease-time 7200;

   subnet 172.16.0.0 netmask 255.255.255.0 {
      option domain-name "blatz";
      option broadcast-address 172.16.0.255;
   }

   host node0 {
      hardware ethernet 00:11:22:33:44:55;
      fixed-address 172.16.0.1;
      next-server 172.16.0.254;
   ```

```
   filename "pxelinux.0";
}

host node0-bmc {
   hardware ethernet 00:11:22:33:44:56;
   fixed-address 172.16.0.101;
}
```

We don't know the real MAC address ("hardware ethernet") for the node yet, but we do know the BMC MAC address. Replace the dummy mac address (00:11:22:33:44:56) with the real MAC address for the BMC of the first node. We will later use the BMC to get the MAC address of this node.

Pay special attention to the lines next−server and filename. next−server is the IP address of a server (the master) that will provide a TFTP service where a PXE file can be downloaded. filename is the name of the file that should be downloaded.

Save the file and exit the editor.

If you copied the example file into place, make sure the ownership of the file is correct. It should be owned by root:root.

3. Validate dhcpd.conf syntax. Any time we configure a service, it's a good idea to verify our configuration. Most services have the ability to confirm that a config is valid. To validate our dhcp config, run:

   ```
   # dhcpd -t
   ```

   If everything is configured correctly, you should get an output like this:

   ```
   Internet Systems Consortium DHCP Server 4.2.5
   Copyright 2004-2013 Internet Systems Consortium.
   All rights reserved.
   For info, please visit https://www.isc.org/software/dhcp/
   Not searching LDAP since ldap-server, ldap-port and ldap-base-dn were
    ↪  not specified in the config file
   ```

   If something is wrong with the file syntax, you will get an output something like this:

   ```
   Internet Systems Consortium DHCP Server 4.2.5
   Copyright 2004-2013 Internet Systems Consortium.
   All rights reserved.
   For info, please visit https://www.isc.org/software/dhcp/
   /etc/dhcp/dhcpd.conf line 6: semicolon expected.
   subnet
    ^
   Configuration file errors encountered -- exiting
   ...
   ```

4. Enable/start dhcpd.

```
# systemctl enable dhcpd
# systemctl start dhcpd
```

5. Verify dhcp status. We can make sure dhcpd start and is running with:

   ```
   # systemctl status dhcpd
   ```

   We should see:

   ```
    dhcpd.service - DHCPv4 Server Daemon
   Loaded: loaded (/usr/lib/systemd/system/dhcpd.service; enabled; vendor
    ↪  preset: disabled)
   Active: active (running) since Wed 2018-04-04 18:31:35 UTC; 7min ago
   ...
   ```

6. Watch the logs to see that our BMC gets its IP address. We can watch the logs with journalctl. We should see that our BMC completes an IPMI handshake. If other nodes are connected, we may see that they attempt to get leases but get a "no leases available" error. This is how it should work since we have not configured those BMCs in the dhcpd.conf file. To watch the logs:

   ```
   # journalctl _SYSTEMD_UNIT=dhcpd.service -f
   ```

   We should eventually see a handshake like this:

   ```
   Apr 02 17:20:52 pxe-master.blah dhcpd[19694]: DHCPDISCOVER from
    ↪  00:11:22:33:44:01 via em1
   Apr 02 17:20:52 pxe-master.blah dhcpd[19694]: DHCPOFFER on
    ↪  176.16.0.101 to 00:11:22:33:44:01 via em1
   Apr 02 17:20:53 pxe-master.blah dhcpd[19694]: DHCPREQUEST for
    ↪  176.16.0.101 (192.168.33.2) from 00:11:22:33:44:01 via em1
   Apr 02 17:20:53 pxe-master.blah dhcpd[19694]: DHCPACK on 176.16.0.101
    ↪  to 00:11:22:33:44:01 via em1
   ```

   Once we see this handshake has completed, we should be able to ping the BMC:

   ```
   # ping 172.16.0.101
   PING localhost (172.16.0.101): 56 data bytes
   64 bytes from 172.16.0.101: icmp_seq=0 ttl=64 time=0.052 ms
   64 bytes from 172.16.0.101: icmp_seq=1 ttl=64 time=0.083 ms
   ```

   DHCP is now functional, and we have an IP address for our BMC.

Step 2: IPMI control of the compute node

IPMI (Intelligent Platform Management Interface) is a fairly universal protocol for interacting with BMCs from various vendors. It can be used to control power, attach to serial consoles, configure some BIOS settings (like boot order), and discover information about the server. We will use it to: collect MAC addresses, control power, and attach to the serial console.

1. Install ipmitool:

   ```
   # yum install ipmitool
   ```

2. Get an IPMI shell. The ipmitool command will allow us to access the features of the BMC available through the IPMI protocol. The general syntax of an IPMI command is:

   ```
   # ipmitool -I lanplus -H <IP address> -U <user> -P <password> <IPMI
    ↪  command>
   ```

   The "lanplus" option specifies a version of the protocol that we will need to use with these Dell servers. There is a special shell IPMI command that will drop us in a shell, where we can run various IPMI commands.

   ```
   # ipmitool -I lanplus -H 172.16.0.101 -U admin -P admin shell
   ```

3. Let's get the MAC addresses of the node:

   ```
   > delloem mac
   ```

   This will give us MAC addresses for all of the ports on this node. We need the MAC of the first interface. Write it down (or copy/paste).

4. *IMPORTANT:* Now that we have the MAC address, we should update the "node0" section of /etc/dhcp/dhcpd.conf before we forget. Edit the file and put this MAC address as "hardware ethernet" for the node0 section. Perform the config check, and restart the dhcpd service:

   ```
   # systemctl restart dhcpd
   ```

   Verify status and look at the log to make sure it is working correctly. We are now done with the dhcpd config.

5. Other useful IPMI commands for later reference:

   - power on/off/cycle - turn power on, off or power cycle the node
   - power status - get current power on/off status
   - sol activate - connect to the serial console. To exit the serial console, use the key sequence <enter>~. (period is part of the sequence).
   - lan print 1 - view the network configuration of the BMC
   - chassis bootdev pxe/disk/bios - set the first boot device to be PXE, disk, or into the BIOS.

Step 3: Configure TFTP

Recall that the "next-server" line of the dhcpd.conf file references a server (master) where a TFTP service can be found. We need to set up that service now. The standard TFTP service is managed through the xinetd service, so we will need to install it as well, and enable/start it. We also need to let xinetd know that it should start TFTP.

1. Install xinetd and tftp (and the tftp client for testing):

```
# yum install xinetd tftp tftp-server
```

2. Tell xinetd to start tftp. Edit the file /etc/xinetd.d/tftp and set the line from disable = yes to disable = no. Notice the line server_args = −s /var/lib/tftpboot. This line tells where TFTP files will be served from. The default is /var/lib/tftpboot, and this is what we will use in this tutorial.

3. Start/enable xinetd

   ```
   # systemctl enable xinetd
   # systemctl start xinetd
   ```

4. Verify and test tftp. We should now see that the xinetd service is active if we run systemctl status xinetd. We can also make sure it is listening for tftp requests. Run:

   ```
   # lsof -i
   COMMAND   PID    USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
   ...
   xinetd    880    root    5u  IPv4  16940      0t0  UDP *:tftp
   ...
   ```

   We should see a line like the above showing the xinetd is, in fact, listening on the UDP tftp port.

   Finally, we can test that tftp is actually working. For this, we need a file in /var/lib/tftpboot. In principle, any file would do.

   ```
   # cp -v /etc/redhat-release /var/lib/tftpboot/
   '/etc/redhat-release' -> '/var/lib/tftpboot/redhat-release'
   # tftp 172.16.0.254
   tftp> get redhat-release
   tftp> quit
   # diff /root/redhat-release /etc/redhat-release
   ```

   We should be able to complete these steps, and see that diff returns no differences between the files.

Step 4: Configure pxelinux

When our node gets its DHCP offer, it will get the next−server``` and ```filename``` options along with it we set was "pxelinux.0". This is a special pxe "kernel" that can serve a purpose similar to grub on a diskfully installed system. It can present a menu of options, download more files from TFTP among other things.

In our setup, we will just have one PXE menu option. It will instruct pxelinux to download a kernel and an initramfs and boot the kernel with some specified options.

1. Install pxelinux:

   ```
   # yum install syslinux-tftpboot
   ```

If you look in /var/lib/tftpboot``` there should now be quite a few files , including ```pxelinux.0.

2. Write the pxelinux configuration. pxelinux.0 will look, through TFTP, in the directory pxelinux.cfg. It will try a sequence of files that map to the node's MAC address or IP address that could load node-specific PXE configurations. If it doesn't find any of these files, it will look for a file called default. We will only use the default file.

   First we need to create the directory:

   ```
   # mkdir /var/lib/tftpboot/pxelinux.cfg
   ```

   Now we need to create the file /var/lib/tftpboot/pxelinux.cfg/default, and place the following contents in the file:

   ```
   DEFAULT menu.c32
   PROMPT 0
   TIMEOUT 50
   SERIAL 0 115200 0
   MENU TITLE Main Menu

   LABEL node
      MENU LABEL Boot x86_64 Compute Node (diskless)
      KERNEL vmlinuz
      APPEND initrd=initramfs.img img=http://172.16.0.254/image.cpio
    ↪  console=tty0 console=ttyS0,115200
   ```

   The KERNEL line says that the kernel to boot is in a file called vmlinuz. We will eventually need that file under /var/lib/tftpboot (Step 7).

   The APPEND line contains kernel parameters. Of special note, the initrd paramters says that we need an initramfs file at /var/lib/tftpboot/initramfs.img. Also, the img parameter is a custom parameter that we will use in our boot process later to know where to download our compute node image. Note: we will need to have an http service for this (Step 6).

   Finally, the SERIAL line and the console=ttyS0... parameters make sure both pxelinux and the kernel know that we want to use a serial port as our console (through the BMC).

3. Testing it out: We don't have everything in place yet to actually get our node to boot, but at this point we can verify that pxelinux works. To do this, I recommend going into split window in tmux (key sequence <ctrl>−b ", use <ctrl>−b <up>/<down> to switch windows). In one window, follow the dhcpd logs with # journalctl _SYSTEMD_UNIT=dhcpd.service −f, and in the other window start an IPMI shell as we did in Step 2. Power on the node (ipmi> power on), wait a few seconds for it to initialize, then attach to the serial console (ipmi> sol activate). You should see the node boot through the BIOS/UEFI, then negotiate DHCP, which will appear in the logs. Then you should see the PXE menu we specified in the serial console. It won't boot beyond this point, because we need the kernel, initramfs and

compute image still.

Step 5: Create our compute image

Next, we need to create the install image that our node is going to run. We are going to make a very minimal image that would be a bit useless for production, but illustrates the basic steps of how this is done.

1. First, we need a directory structure where we will put the image contents. Make the directory /opt/img/root. This will be the root of our compute node's filesystem. Under /opt/img/root create an etc directory. There is a yum.conf file in the /root/netboot directory. Copy that file into /opt/img/root/etc.

2. Now we can use the −−installroot feature of yum to install the set of packages required for a "Minimal Install" of CentOS. We need to use the repositories defined in our yum.conf file we just copied in. To do that we also need to make sure the default repositories on the master are disabled. The repos in the yum.conf are conveniently named [img-base] and [img-updates] (it's not a bad idea to take a look at the contents of the yum.conf). This can all be accomplished with the following yum command (*this will take several minutes to complete*):

   ```
   # yum --installroot=/opt/img/root --disablerepo='*'
    ↪  --enablerepo='img-*' groupinstall "Minimal Install"
   ```

   Once this has finished, you should have a fairly complete looking Linux install in /opt/img/root.

3. There are several configuration files that need to be set up in our image file. In the interest of brevity, I have pre-made these files so they can be copied into the image. We first make sure the file ownership is correct before moving them into place.

   ```
   # chown -R root:root /root/netboot/root
   # cp -av /root/netboot/root/* /opt/img/root/
   ```

   There are a couple of files of particular interest:

   - /etc/ssh/sshd_config - contains the configuration for the sshd service that will run on our node. Specifically, this is altered to allow root to log in with a public key.
   - /init - *IMPORTANT!* CentOS 7 systems use systemd as their init process; however, systemd requires more work to get to start correctly than we are going to bother with here. This /init is a short script that will handle the couple of basic bootup tasks we need by hand, such as starting sshd and agetty (login services). Take a look inside this script. Once the image is loaded, it will run this script as Process ID (PID) 1.

4. To set up public key authentication for our node, we need to copy a public key from the master to the image. First, generate a public key on the master (if you haven't

already):

```
# ssh-keygen
```

And just hit Enter for all of the prompts. This will create /root/.ssh/{id_rsa,id_rsa.pub} on the master. Now, we need to copy this file to a special location in the image: /opt/img/root/root/.ssh/authorized_keys, and make sure all permissions are correct:

```
# mkdir /opt/img/root/root/.ssh
# chmod 700 /opt/img/root/root/.ssh
# cp -v /root/.ssh/id_rsa.pub /opt/img/root/root/.ssh/authorized_keys
# chmod 600 /opt/img/root/root/.ssh/authorized_keys
```

IMPORTANT! if permissions are incorrect on these files and directories, public key authentication will not work.

Our image is now ready. All that remains is to package it up and make it available through http, which we will do in the next step.

Step 6: Setup httpd and publish image

We will not require much from http. It just needs to send a single file. We will use the default configuration, which makes things easy.

1. Install, enable and start httpd:

   ```
   # yum install httpd
   # systemctl enable httpd
   # systemctl start httpd
   ```

2. Verify the httpd service. We can use the tools we used before to verify services. systemctl status httpd should return an "active" status. lsof –i should show that httpd is listening on TCP on "*:http". Finally, Apache comes with a default page when installed and started. We can load that page in text mode with: links http://172.16.0.254. This should load a"welcome to Apache" page.

3. Bundling & publishing our compute image: Apache's default DocumentRoot is /var/www/html. We want to bundle up a version of our image in /opt/img/root that can be easily downloaded, and place it in that directory so Apache will serve the file. We will use the cpio file format to store our image. In a larger cluster, we might also compress it. The following commands will publish our image (note: it's important to start in the root directory of the image):

   ```
   # cd /opt/img/root
   # find . | cpio -oc > /var/www/html/image.cpio
   ```

   Cpio will print a message saying how many bytes it wrote. You should be able to see the file at /var/www/html/image.cpio. As a test, # wget http://172.16.0.254/image.cpio should download the file.

Step 7: Bootstrap (kernel & initramfs)

The final piece to get our node to boot is the most complicated. It involves some custom scripting to build an initramfs that will:

1. Load network device modules
2. Get an IP from the DHCP server
3. Setup a ramfs mount for our real root
4. Download and extract the compute image
5. switch_root to the new image and start the init in the image

. . . all while doing various system tasks, like mounting /proc.

Rather than have you actually go through this process, which would be a whole session, we have created a script that will generate the initramfs for you and copy a vmlinuz and initramfs.img file into /var/lib/tftpboot where they can be downloaded through TFTP.

To use this script:

```
# cd /root/netboot
# sh bootstrap.sh
```

Verify that the vmlinuz and initramfs.img files are in /var/lib/tftpboot.

Finally, we should at least get a sense of how things are done. Take a look at the bootstrap.sh script to see what it does.

To illustrate what boostrap made for us, do the following:

```
# cd /var/lib/tftpboot
# mkdir initramfs
# cd initramfs
# cpio -iv < ../initramfs.img
```

This will extract the initramfs image into the directory /var/lib/tftpboot/initramfs. There are some important files here:

- /init - this process is the first to launch. It manages everything else and ultimately performs the switch_root.
- /load_img.sh - this helper script is what downloads and extracts the compute image. It also extracts the img= kernel command line argument to know where to get the image.
- /modules.txt - this file lists modules that need to be loaded at this stage (mostly network device drivers).

There are a couple of other small helper scripts in this image as well. You are strongly encouraged to read through these files, especially /init and /load_img.sh to see how this process works.

Step 8: Boot the compute node

All of the pieces should be in place now! Go ahead and attach to the serial console and power cycle the node. You should see it go through these stages:

1. BIOS/UEFI init
2. pxelinux menu (times out in 5 seconds)
3. the kernel will load
4. the initramfs will load, and image will download
5. it will switch to the real root, load a couple of services, and give a login prompt

At this point, the node is up. We should be able to ssh to it using: ssh 172.16.0.1, and (after accepting the host key) it should public key authenticate.

That's it!

Building a cluster with OpenHPC & Warewulf

Overview

Step 0: Base install of the master

We want to start our master from a clean slate. We will need to redo the following steps to get going:

1. Re-install the master so we're starting from a clean slate. Go ahead and set up the network for both em1 and em2.
2. Set your hostname: hostnamectl set−hostname <hostname>
3. Re-add our users.
4. Disable selinux (setenforce 0, and edit /etc/selinux/config)
5. Disable firewalld (systemctl stop firewalld, systemctl disable firewalld)
6. Configure yum repos (found at http://10.0.52.146/repo/yum.repos.d.tar.gz)
7. [root@master]# yum −y install tmux vim ntp

Don't worry, this will be the last time we have to do these steps by hand.

Step 1: Install OpenHPC components

Verify OpenHPC repo setup

Ordinarily, you would have to set up access to the *OpenHPC* repository. Fortunately, we've had it all along from our local mirror list!

To verify this:

```
[root@master ~]# yum repolist
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
repo id             repo name           status
OpenHPC             OpenHPC             327
OpenHPC-updates     OpenHPC Updates     739
base                CentOS - Base       10,019
epel                EPEL                13,578
extras              CentOS - Extras     260
updates             CentOS - Updates    994
repolist: 25,917
```

Note OpenHPC and OpenHPC−updates. Also, a look at /etc/yum.repos.d/OpenHPC.repo to verify that the repository is configured and enabled.

To see what packages are available from OpenHPC, you can use the following:

```
[root@master ~]# yum --disablerepo='*' --enablerepo='OpenHPC*' list
 ↪  available
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Available Packages
EasyBuild-ohpc.x86_64       3.8.1-5.1.ohpc.1.3.7    OpenHPC-updates
R-gnu7-ohpc.x86_64          3.5.0-2.1               OpenHPC-updates
R-gnu8-ohpc.x86_64          3.5.2-4.1.ohpc.1.3.7    OpenHPC-updates
R_base-ohpc.x86_64          3.3.3-22.3              OpenHPC-updates
adios-gnu-impi-ohpc.x86_64  1.12.0-10.1             OpenHPC-updates
...
```

As you can see, OpenHPC provides quite a few (~1000) useful packages for OpenHPC. We'll be using quite a few of these.

Installing OpenHPC packages we'll need

To get started, we'll install the OpenHPC base packages, and Warewulf itself:

```
[root@master ~]# yum -y install ohpc-base ohpc-warewulf
```

This will install quite a few packages (>150). Note: the −y option tells yum to just install the packages, don't prompt us to verify that we want to install them.

Step 2: Initial Warewulf setup

Setup Warewulf `provision.conf`

We need to edit the file /etc/warewulf/provision.conf to set some basic parameters about our cluster. Specifically, we need to set device = em1. Your provision.conf file should look like this:

```
# What is the default network device that the master will use to
# communicate with the nodes?
network device = em1

# Which DHCP server implementation should be used?
dhcp server = isc

# What is the TFTP root directory that should be used to store the
# network boot images? By default Warewulf will try and find the
# proper directory. Just add this if it can't locate it.
#tftpdir = /var/lib/tftpboot

# Automatically generate and manage a dynamnic_host virtual file
# object in the datastore? This is useful for provisioning this
# out to nodes so they always have a current /etc/hosts file.
generate dynamic_hosts = yes

# Should we manage and overwrite the local hostfile file on this
# system? This will cause all node entries to be added
# automatically to /etc/hosts.
update hostfile = yes

# If no cluster/domain is set on a node, should we add 'localdomain'
# as the default domain
use localdomain = yes

# The default kernel arguments to pass to the nodes boot kernel
default kargs = "net.ifnames=0 biosdevname=0 quiet"
```

Initialize Warewulf

Warewulf uses a database (mariadb) as its backend store for information about your cluster. This has already been installed as a dependency for ohpc−warewulf but we need to tell warewulf to initialize the database tables it needs. Run:

```
[root@master ~]# wwinit database
database:     Checking to see if RPM 'mysql-server' is installed
 ↪   NO
database:     Checking to see if RPM 'mariadb-server' is installed
 ↪   OK
database:     Activating Systemd unit: mariadb
database:      + /bin/systemctl -q enable mariadb.service
 ↪   OK
database:      + /bin/systemctl -q restart mariadb.service
 ↪   OK
database:      + mysqladmin
 ↪  --defaults-extra-file=/tmp/0.6cgpUIwx0LMt/my.cnf OK
database:     Database version: UNDEF (need to create database)
database:      + mysql --defaults-extra-file=/tmp/0.6cgpUIwx0LMt/my.cnf
 ↪  ware OK
database:      + mysql --defaults-extra-file=/tmp/0.6cgpUIwx0LMt/my.cnf
 ↪  ware OK
database:      + mysql --defaults-extra-file=/tmp/0.6cgpUIwx0LMt/my.cnf
 ↪  ware OK
database:     Checking binstore kind
 ↪  SUCCESS
Done.
```

Warewulf will use ssh keys to provide access to nodes, just like we did in the netboot tutorial. It will manage this for us, but we have to initialize them:

```
[root@master ~]# wwinit ssh_keys
ssh_keys:     Checking ssh keys for root
 ↪   NO
ssh_keys:     Generating ssh keypairs for local cluster access:
ssh_keys:      + ssh-keygen -t rsa -f /root/.ssh/cluster -N
 ↪   OK
ssh_keys:     Updating authorized keys
 ↪   OK
ssh_keys:     Checking root's ssh config
 ↪   NO
ssh_keys:     Creating ssh configuration for root
 ↪  DONE
ssh_keys:     Checking for default RSA host key for nodes
 ↪   NO
ssh_keys:     Creating default node ssh_host_rsa_key:
ssh_keys:      + ssh-keygen -q -t rsa -f
 ↪  /etc/warewulf/vnfs/ssh/ssh_host_rsa OK
ssh_keys:     Checking for default DSA host key for nodes
 ↪   NO
ssh_keys:     Creating default node ssh_host_dsa_key:
```

```
ssh_keys:      + ssh-keygen -q -t dsa -f
 ↪  /etc/warewulf/vnfs/ssh/ssh_host_dsa OK
ssh_keys:     Checking for default ECDSA host key for nodes
 ↪   NO
ssh_keys:     Creating default node ssh_host_ecdsa_key:

 ↪   OK
ssh_keys:     Checking for default Ed25519 host key for nodes
 ↪   NO
ssh_keys:     Creating default node ssh_host_ed25519_key:

 ↪   OK
Done.
```

It's worth looking through the output of this command to see exactly what it did. Where did it put the keys it generated?

Setup NFS exports

We will need a couple of shared filesystems for our cluster to work properly. We will use the Network File System (NFS) for this. NFS will let our nodes easily attach to directories on the master over our Cluster LAN network. Our nodes will need to mount /home so that our users can have their home directories on all of the nodes. We will also need /opt/ohpc/pub, for sharing things like add-on software with our nodes.

To share a filesystem via NFS, we need to add entries to /etc/exports. The format of this file is:

```
<directory_to_share>     <network|host|*>(<options>)
```

Where <directory_to_share> is the directory we want to give access to (e.g. /home), <network|host|*> is either a network spec (e.g. 172.16.0.0/24), a specific host by IP or hostname, or * to indicate *any* host is allowed to mount the NFS share. (<options>) specifies special options we want for our share. Some common options are:

- rw or ro to indicate "read-write" or "read-only"
- root_squash or no_root_squash: root_squash indicates that the root user on the remote system should be treated as the nobody user, i.e. have no special permissions on the share. This is usually a good idea for security.
- no_subtree_check disables an expensive check that NFS does by default. It's common to use this option. For details, see: man 5 exports
- fsid=<num> sets a unique identifier for each mountpoint.

Our /etc/exports file should look like:

```
/home           172.16.0.0/24(rw,no_subtree_check,fsid=10,root_squash)
/opt/ohpc/pub   172.16.0.0/24(ro,no_subtree_check,fsid=11)
```

Note that /opt/ohpc/pub doesn't have the root_squash option. Since this is exported ro, root on a remote system won't be able to modify anything here, but we do need to make sure that root on the remote system can *read* everything we share here in order to use some of the software that will be provided from this share.

Now we need to actually enable the service that will make these shares available.

```
[root@master ~]# systemctl start nfs-server
[root@master ~]# systemctl enable nfs-server
Created symlink from
 ↪  /etc/systemd/system/multi-user.target.wants/nfs-server.service to
 ↪  /usr/lib/systemd/system/nfs-server.service.
```

If you change /etc/exports while NFS is running, you will need to tell NFS to re-read the file. You can either do this by restarting nfs−server, or by running:

```
[root@master ~]# exportfs -a
```

This will also check the syntax of your exports file before changing anything, which makes it a bit safer than restarting nfs−server.

We can use the showmount command to get information on who is connected to a share, as well as what is being exported by a host (including ourselves). To see what is being exported, run:

```
[root@master ~]# showmount -e
Export list for master:
/opt/ohpc/pub 172.16.0.0/24
/home         172.16.0.0/24
```

To see clients mounting our shares, run showmount with no options:

```
[root@master ~]# showmount
Hosts on master:
```

Of course, we haven't mounted the share yet, so this is empty.

To mount an NFS share, we can either make an entry in fstab (we'll do this later), or mount by hand with (we can mount our own NFS volume to verify):

```
[root@master ~]# mount -t nfs 172.16.0.254:/home /mnt
[root@master ~]# grep nfs /proc/self/mounts
172.16.0.254:/home /mnt nfs4 rw,relatime,vers=4.1,rsize=1048576,wsize=10485⌋
 ↪  76,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,clientaddr=172⌋
 ↪  .16.0.254,local_lock=none,addr=172.16.0.254 0
 ↪  0
[root@master ~]# ls /mnt
lowell
[root@master ~]# umount /mnt
```

This mounted /home on 172.16.0.254 onto our local directory of /mnt.

Our NFS shares are now ready.

Setup necessary services

We need to also set up the following services (we've set up all of these before, so we won't go into detail):

1. NTP:
   1. Remove all current lines starting with server from /etc/ntp.conf
   2. Add a line with server 172.16.0.146
   3. systemctl enable ntpd
   4. systemctl start ntpd
   5. Verify with ntpq then lpeers
2. TFTP:
   1. Edit /etc/xinetd.d/tftp, change disabled = yes to disabled = no
   2. systemctl enable xinetd
   3. systemctl start xinetd
   4. Verify this by downloading a file from /var/lib/tftpboot with the tftp client (tftp may need to be installed).
3. HTTPD (Apache):
   1. systemctl enable httpd
   2. systemctl start httpd
   3. Verify this by running wget http://localhost/ and making sure you get an HTML response. This may be an error response (e.g. 403); that's fine, as long as it gets something (wget may need to be installed).
4. *Enable* (but don't start) DHCPD. DHCPD isn't configured yet; warewulf will do this for us later. But, we want to make sure it's enabled.
   1. systemctl enable dhcpd

Step 3: Building the BOS

In this step, we'll build the Base Operating System (BOS). This will be what our node image, called the VNFS, is made from. The steps are similar to what we did in the netboot guide, but warewulf provides some tools to help us out.

Building the initial chroot

We can build out our base image with a single warewulf command. Note: this will take a little while:

```
[root@master ~]# wwmkchroot centos-7 /opt/ohpc/admin/images/centos7
Loaded plugins: fastestmirror
Determining fastest mirrors
os-base                                | 3.6 kB  00:00:00
(1/2): os-base/x86_64/group_gz         | 166 kB  00:00:00
(2/2): os-base/x86_64/primary_db       | 6.0 MB  00:00:00
Resolving Dependencies
--> Running transaction check
... (lots of package installs)

Complete!
```

If we look in /opt/ohpc/admin/images/centos7 we'll see there is a root filesystem there now:

```
[root@master ~]# ls /opt/ohpc/admin/images/centos7/
bin  boot  dev  etc  fastboot  home  lib  lib64  media  mnt  opt  proc
 ↪  root  run  sbin  srv  sys  tmp  usr  var
```

That's it. We now have the base of the image. If we inspect the image, we'll notice that warewulf works hard to keep this image small:

```
[root@master ~]# du -sh /opt/ohpc/admin/images/centos7
454M    /opt/ohpc/admin/images/centos7
```

Warewulf does this by leaving out a lot of software you'd ordinarily want. For instance, even yum isn't installed in the BOS, because we anticipate that all installs would happen on the master when we create the image.

Adding software to the BOS

We need to add a little more software to our master. In general, we just add software to the BOS using yum with the −−installroot= option.

```
[root@master ~]# yum -y --installroot=/opt/ohpc/admin/images/centos7
 ↪  install ohpc-base-compute ntp kernel ipmitool lmod-ohpc
Loaded plugins: fastestmirror
Determining fastest mirrors
OpenHPC                | 2.9 kB  00:00:00
OpenHPC-updates        | 2.9 kB  00:00:00
base                   | 3.6 kB  00:00:00
epel                   | 3.6 kB  00:00:00
extras                 | 2.9 kB  00:00:00
updates                | 2.9 kB  00:00:00
Resolving Dependencies
... (installs a bunch of packages)

Complete!
```

Adding our NFS mounts to the BOS image

We need to add fstab entries so that our NFS shares get mounted when our nodes boot. Open /opt/ohpc/admin/images/centos7/etc/fstab and add these lines to the bottom of the file:

```
172.16.0.254:/home /home nfs nfsvers=3,nodev,nosuid,noatime 0 0
172.16.0.254:/opt/ohpc/pub /opt/ohpc/pub nfs nfsvers=3,nodev,noatime 0 0
```

These specify our NFS mounts as default mounts to add at startup.

Similar changes can be made to other configuration files in the image (this is all we need for now), but we will see in a moment that warewulf has a feature that makes managing some of these files easier.

Enabling services in the BOS image

We need to enable the NTPD service *inside* our BOS. Fortunately, systemctl has a −−root= option. We can do this with:

```
[root@master ~]# systemctl --root=/opt/ohpc/admin/images/centos7 enable
 ↪  ntpd
Created symlink /opt/ohpc/admin/images/centos7/etc/systemd/system/multi-use⌋
 ↪  r.target.wants/ntpd.service, pointing to
 ↪  /usr/lib/systemd/system/ntpd.service.
```

This will make sure NTPD starts on our compute nodes.

NTPD is the only service we need to enable for the moment.

Importing files into Warewulf images

Warewulf has a feature that allows the synchronization of files from the master into the image. This is especially useful for things like the passwd file that we would like to update easily when we want to add a user.

This is achieved with the wwsh file import command. Note that wwsh on our own would drop us into a special shell where we could enter various warewulf commands. Try wwsh help to get a sense of what is available.

Let’s import some useful files:

```
[root@master ~]# wwsh file import /etc/passwd
[root@master ~]# wwsh file import /etc/group
[root@master ~]# wwsh file import /etc/shadow
```

We can see what we have imported with:

```
[root@master ~]# wwsh file list
group                   :  rw-r--r-- 1   root root              619
 ↪  /etc/group
passwd                  :  rw-r--r-- 1   root root             1370
 ↪  /etc/passwd
shadow                  :  rw-r----- 1   root root              890
 ↪  /etc/shadow
```

Note that warewulf also keeps track of the correct ownership and permissions for the files.

If we ever change one of these files, we need to re-sync them with warewulf. We can do this with:

```
[root@master ~]# wwsh file resync
```

Our BOS is now complete.

Step 4: Assembling Bootstrap/VNFS & adding the first 3 nodes

Just like we did in the netboot example, warewulf needs to assemble a "bootstrap" (i.e. a kernel and initramfs) as well as make a useable version of the BOS image (we did this with cpio).

Assembling the VNFS

Warewulf calls the packaged image a VNFS (Virtual Node File System). A Warewulf cluster could potentially have many different images available, and it maintains a mapping that keeps track of which nodes should use which VNFSes.

The VNFS is stored in the mariadb database that keeps stores all of the warewulf configuration information. We create and import our BOS as a VNFS with a single command:

```
[root@master ~]# wwvnfs --chroot=/opt/ohpc/admin/images/centos7
 ↪  centos7-base
Creating VNFS image from centos7-base
Compiling hybridization link tree                              : 0.13 s
Building file list                                             : 0.40 s
Compiling and compressing VNFS                                 : 11.88 s
Adding image to datastore                                      : 31.97 s
Wrote a new configuration file at: /etc/warewulf/vnfs/centos7-base.conf
Total elapsed time                                             : 44.37 s
```

Here, −−chroot specified where or BOS lives. The last argument, centos7−base specifies the name of our VNFS. This argument is optional. Had we left it off, our VNFS gets named the same name as the directory, in our case, centos7.

We can use wwsh to show that we have successfully imported the image, and get some information about it:

```
[root@master ~]# wwsh vnfs list
VNFS NAME            SIZE (M)   ARCH       CHROOT LOCATION
centos7-base         261.2      x86_64     /opt/ohpc/admin/images/centos7
```

Warewulf now knows how to use the BOS image we built.

Assembling the bootstrap

We now need the intramfs and kernel that will want to use. Just like we did with the netboot tutorial, the initramfs may need some extra pieces like extra kernel modules. Warewulf uses the file /etc/warewulf/bootstrap.conf to configure this. We will want to add the following line to this file:

```
drivers += updates/kernel/
```

This can safely be added to the end of the file.

Warewulf gives us a single command to build and assemble the bootstrap:

```
[root@master ~]# wwbootstrap $(uname -r)
Number of drivers included in bootstrap: 541
Number of firmware images included in bootstrap: 96
Building and compressing bootstrap
Integrating the Warewulf bootstrap: 3.10.0-957.12.2.el7.x86_64
Including capability: provision-adhoc
Including capability: provision-files
Including capability: provision-selinux
Including capability: provision-vnfs
Including capability: setup-filesystems
Including capability: setup-ipmi
Including capability: transport-http
Compressing the initramfs
Locating the kernel object
Bootstrap image '3.10.0-957.12.2.el7.x86_64' is ready
Done.
```

The only argument we need to wwbootstrap is the kernel version we want. We've taken a shortcut here by using uname −r (try it, it prints our current kernel version) because we know that the kernel version we are running on the master is the same as the kernel version we want on our nodes. This is not necessarily always the case.

Like the VNFS, we can view information about our bootstrap with wwsh:

```
[root@master ~]# wwsh bootstrap list
BOOTSTRAP NAME            SIZE (M)      ARCH
```

```
3.10.0-957.12.2.el7.x86_64 29.1          x86_64
```

Adding the first three compute nodes

We already have the MAC address information for the first three compute nodes, so we can add them to the warewulf configuration. We will get the rest of the nodes added in the next step.

We add nodes with the wwsh node new command. It takes several arguments to fully specify a node:

```
[root@master ~]# wwsh node new n01 --ipaddr=172.16.0.1
 ↪  --netmask=255.255.255.0 --gateway=172.16.0.254
 ↪  --hwaddr=18:66:da:ea:34:7c -D eth0 -g compute
Are you sure you want to make the following 7 change(s) to 1 node(s):

     NEW: NODE                 = n01
     SET: eth0.HWADDR          = 18:66:da:ea:34:7c
     SET: eth0.IPADDR          = 172.16.0.1
     SET: eth0.NETMASK         = 255.255.255.0
     SET: eth0.GATEWAY         = 172.16.0.254
     SET: GROUPS               = compute

Yes/No [no]> Yes
```

These arguments should be mostly self-explanatory. The −g option allows you to add the node to a “group.” Warewulf groups allow you to apply certain kinds of configuration to all nodes in the group or act on multiple nodes at once.

Go ahead and add the other two nodes now in the same way.

Once you’re done, you can list your nodes:

```
[root@master ~]# wwsh node list
NAME                GROUPS              IPADDR              HWADDR
============================================================================↩
 ↪  ==
n01                 compute             172.16.0.1
 ↪  18:66:da:ea:34:7c
n02                 compute             172.16.0.2
 ↪  18:66:da:ea:23:d8
n03                 compute             172.16.0.3
 ↪  18:66:da:ea:4a:c8
```

You can get more information by using wwsh node print:

```
[root@master ~]# wwsh node print n01
```

```
#### n01
↪ ##########################################################################
            n01: ID                  = 7
            n01: NAME                = n01
            n01: NODENAME            = n01
            n01: ARCH                = x86_64
            n01: CLUSTER             = UNDEF
            n01: DOMAIN              = UNDEF
            n01: GROUPS              = compute
            n01: ENABLED             = TRUE
            n01: eth0.HWADDR         = 18:66:da:ea:34:7c
            n01: eth0.HWPREFIX       = UNDEF
            n01: eth0.IPADDR         = 172.16.0.1
            n01: eth0.NETMASK        = 255.255.255.0
            n01: eth0.NETWORK        = UNDEF
            n01: eth0.GATEWAY        = 172.16.0.254
            n01: eth0.MTU            = UNDEF
```

We've added our nodes, but we haven't yet said which VNFS or bootstrap they should use. This is handled using the wwsh provision command. We can see what we have now:

```
[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP              FILES
============================================================================
↪ ==
n01                 UNDEF           UNDEF                  group,passwd
n02                 UNDEF           UNDEF
↪ dynamic_hosts,grou...
n03                 UNDEF           UNDEF
↪ dynamic_hosts,grou...
```

We can set their provision information and use the "compute" group we created to do them all at once:

```
[root@master ~]# wwsh provision set n[01-03] --vnfs=centos7-base
↪ --bootstrap=$(uname -r) --console="ttyS0,115200"
↪ --files=dynamic_hosts,passwd,group,shadow
Are you sure you want to make the following changes to 3 node(s):

     SET: BOOTSTRAP             = 3.10.0-957.12.2.el7.x86_64
     SET: VNFS                  = centos7-base
     SET: FILES                 = dynamic_hosts,passwd,group,shadow
     SET: CONSOLE               = ttyS0,115200

Yes/No> Yes
```

Notice that we set the VNFS and bootstrap, but we also set −−console and −−files.

−−console sets our kernel command line parameters to include the serial console. −−files attaches our imported files to these nodes.

Let's get the provision list again:

```
[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP             FILES
==========================================================================
↪ =====
n01                 centos7-base    3.10.0-957.12.2.el...
↪ dynamic_hosts,grou...
n02                 centos7-base    3.10.0-957.12.2.el...
↪ dynamic_hosts,grou...
n03                 centos7-base    3.10.0-957.12.2.el...
↪ dynamic_hosts,grou...
```

That looks better.

Let's get a little more detail with wwsh provision print:

```
[root@master ~]# wwsh provision print n01
#### n01
↪ #########################################################################
            n01: BOOTSTRAP        = 3.10.0-957.12.2.el7.x86_64
            n01: VNFS             = centos7-base
            n01: FILES            = dynamic_hosts,group,passwd,shadow
            n01: PRESHELL         = FALSE
            n01: POSTSHELL        = FALSE
            n01: CONSOLE          = ttyS0,115200
            n01: PXELINUX         = UNDEF
            n01: SELINUX          = DISABLED
            n01: KARGS            = "net.ifnames=0 biosdevname=0 quiet"
            n01: BOOTLOCAL        = FALSE
```

This shows us our full file list and the console setting.

Note the file "dynamic_hosts". This is a special file that warewulf will auto-generate for us. It will create an /etc/hosts file that lists entries for all known nodes automatically. It uses /etc/hosts on the master as a base template to build off of.

Getting access to the BMCs

You'll note that the warewulf config did not include the BMCs. Warewulf will be generating our dhcpd.conf file for us, so we need a place to put some static entries. Fortunately, warewulf has a mechanism for this. We can modify the file /etc/warewulf/dhcpd−template.conf to add our BMC entries. We need to add them *before* the line that reads # Node entries will follow below. Our file should look like:

```
...

host n01-bmc { hardware ethernet 18:66:da:68:3e:40; fixed-address
 ↪  172.16.0.101; }
host n02-bmc { hardware ethernet 18:66:da:68:4b:14; fixed-address
 ↪  172.16.0.102; }
host n03-bmc { hardware ethernet 18:66:da:68:41:3a; fixed-address
 ↪  172.16.0.103; }


# Node entries will follow below
```

Finishing it up

To generate the dhcpd config, we run:

```
[root@master warewulf]# wwsh dhcp update
Rebuilding the DHCP configuration
Done.
```

We'll need to do this any time we make changes to dhcpd−template.conf. Let's take a look at /etc/dhcp/dhcpd.conf now. We'll see things that look like this:

```
host n01-bmc { hardware ethernet 18:66:da:68:3e:40; fixed-address
 ↪  172.16.0.101; }
host n02-bmc { hardware ethernet 18:66:da:68:4b:14; fixed-address
 ↪  172.16.0.102; }
host n03-bmc { hardware ethernet 18:66:da:68:41:3a; fixed-address
 ↪  172.16.0.103; }

# Node entries will follow below


group {
   # Evaluating Warewulf node: n01 (DB ID:7)
   # Adding host entry for n01-eth0
   host n01-eth0 {
      option host-name n01;
      option routers 172.16.0.254;
      hardware ethernet 18:66:da:ea:34:7c;
      fixed-address 172.16.0.1;
      next-server 172.16.0.254;
   }
```

While a bit more complicated than what we did in the netboot tutorial, this should look very familiar.

Just like we did in the netboot tutorial, verify the config with dhcpd −t, and (re)start the dhcpd service:

```
[root@master warewulf]# dhcpd -t
Internet Systems Consortium DHCP Server 4.2.5
Copyright 2004-2013 Internet Systems Consortium.
All rights reserved.
For info, please visit https://www.isc.org/software/dhcp/
Not searching LDAP since ldap-server, ldap-port and ldap-base-dn were not
 ↪  specified in the config file
```

```
[root@master warewulf]# systemctl restart dhcpd
```

```
[root@master warewulf]# systemctl status dhcpd
 dhcpd.service - DHCPv4 Server Daemon
   Loaded: loaded (/usr/lib/systemd/system/dhcpd.service; disabled; vendor
 ↪  preset: disabled)
   Active: active (running) since Sat 2019-06-08 13:54:43 MDT; 3s ago
     Docs: man:dhcpd(8)
           man:dhcpd.conf(5)
 Main PID: 31440 (dhcpd)
   Status: "Dispatching packets..."
   CGroup: /system.slice/dhcpd.service
           31440 /usr/sbin/dhcpd -f -cf /etc/dhcp/dhcpd.conf -user dhcpd
 ↪  -group dhcpd --no-pid

Jun 08 13:54:43 master systemd[1]: Started DHCPv4 Server Daemon.
... (some log entries)
```

To generate the pxelinux.cfg configs, we run:

```
[root@master warewulf]# wwsh pxe update
```

Booting our first three nodes

At this point (it may take a bit for DHCP to assign addresses), we should be able to reboot our nodes. Let's just do the first one to start:

```
[root@master warewulf]# ipmitool -I lanplus -H 172.16.0.101 -U admin -P
 ↪  admin shell
ipmitool> power status
Chassis Power is on
ipmitool> power off
Chassis Power Control: Down/Off
ipmitool> power status
Chassis Power is off
ipmitool> chassis bootdev pxe options=persistent
```

```
Set Boot Device to pxe
ipmitool> power on
Chassis Power Control: Up/On
ipmitool> sol activate
[SOL Session operational.  Use ~? for help]
...
```

We should be able to watch the node fully boot.

We'll know warewulf is working when we see a screen like:

```
Now Booting Warewulf...

Setting the hostname (n01):
 ↪   OK
Loading drivers: uhci-hcd ohci-hcd ehci-hcd whci-hcd isp116x-hcd
 ↪  isp1362-hcd OKci-hcd sl811-hcd sd_mod
Detecting hardware: ahci ahci tg3 tg3 tg3 tg3 megaraid_sas mlx5_core
 ↪   OK
Bringing up local loopback network:
 ↪   OK
Checking for network device: eth0 (eth0)
 ↪   OK
Configuring eth0 (eth0) statically: (172.16.0.1/255.255.255.0)
 ↪   OK
Configuring gateway: (172.16.0.254)
 ↪   OK
Creating network initialization files: (eth0)
 ↪   OK
Trying to reach the master node at 172.16.0.254 .
 ↪   OK
Probing for HW Address: (18:66:da:ea:34:7c)
 ↪   OK
Starting syslogd:
 ↪   OK
Getting base node configuration:
 ↪   OK
Starting the provision handler:
 * adhoc-pre
 ↪   OK
 * ipmiconfig Auto configuration not activated
 ↪  SKIPPED
 * filesystems
 ↪  RUNNING
   * mounting /
 ↪   OK
 * filesystems
↪   OK
 * getvnfs
↪  RUNNING
   * fetching centos7-base (ID:4)
```

After a couple of minutes we should get a login prompt:

```
CentOS Linux 7 (Core)
Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64

n01 login:
```

To exit the sol session, hit <enter> ~ ~ .

Now:

1. verify that you can SSH to n01 from the master
2. boot the other two nodes!

Step 5: Discovering your nodes (requires physical access)

We don't know the MAC addresses of nodes 3-10. Warewulf provides a tool we can use to discover these and add them to the node list all in one step called wwnodescan. We will have to have physical access to our systems to do this since we also don't have their BMCs configured.

wwnodescan works by listening for unknown MAC addresses and automatically adding a new node for each one it sees.

Because we also have the BMCs on the same network ***we must disconnect the BMCs (purple ethernet)*** through this procedure. We'll re-connect them when we're done. If we don't do this, warewulf will try to add the BMC addresses as new nodes, and we don't want this.

Our general procedure is:

1. start wwnodescan with appropriate options
2. power on a node
3. wait for warewulf to register it
4. power on the next node
5. repeat until all nodes are added

This process is a little tedious, but it's so much easier than collecting MAC addresses by hand!

Before we go to the server room, we'll want to set up what the default configuration for a new node is. We do this by adding a special node called "DEFAULT":

```
[root@master etc]# wwsh node new DEFAULT --groups=compute
Are you sure you want to make the following 3 change(s) to 1 node(s):

     NEW: NODE                     = DEFAULT
     SET: GROUPS                   = compute

Yes/No [no]> Yes
```

Now we can set what the default provision settings are by "provisioning" the DEFAULT node:

```
[root@master etc]# wwsh provision set DEFAULT --vnfs=centos7-base
 ↪  --bootstrap=$(uname -r) --files=dynamic_hosts,passwd,shadow,group
 ↪  --console=ttyS0,115200
Are you sure you want to make the following changes to 1 node(s):

     SET: BOOTSTRAP                = 3.10.0-957.12.2.el7.x86_64
     SET: VNFS                     = centos7-base
     SET: FILES                    = dynamic_hosts,passwd,shadow,group
     SET: CONSOLE                  = ttyS0,115200

Yes/No> Yes
```

We can verify with:

```
[root@master ~]# wwsh provision print DEFAULT
#### DEFAULT
 ↪  ####################################################################
        DEFAULT: BOOTSTRAP         = 3.10.0-957.12.2.el7.x86_64
        DEFAULT: VNFS              = centos7-base
        DEFAULT: FILES             = dynamic_hosts,group,passwd,shadow
        DEFAULT: PRESHELL          = FALSE
        DEFAULT: POSTSHELL         = FALSE
        DEFAULT: CONSOLE           = ttyS0,115200
        DEFAULT: PXELINUX          = UNDEF
        DEFAULT: SELINUX           = DISABLED
        DEFAULT: KARGS             = "net.ifnames=0 biosdevname=0 quiet"
        DEFAULT: BOOTLOCAL         = FALSE
```

Here are the steps in detail (performed in the server room):

1. Make sure all of nodes n[04−10] are powered off
2. Disconnect the BMCs (purple ethernet)
3. Start wwnodescan. We want to give it options to set everything we need up from the beginning. Because we set defaults, we don't need to give provisioning options to wwnodescan. Our command looks like (it's recommended to run this inside a tmux that can be attached to):

```
[root@master ~]# wwnodescan -v --ipaddr=172.16.0.4
 ↪  --netmask=255.255.255.0 --listen=em1 n[04-10]
Successfully connected to database!
 Assuming the nodes are booting over eth0Listening on em1 for DHCP
 ↪  requests
 Scanning for node(s) (Ctrl-C to exit)...
```

4. Power on node 4 (with the power button)
5. Wait a couple of minutes for the node to try to PXE boot. When it tries, you'll see warewulf add it to the list:

   ```
   WARNING:  Auto-detected "n04", cluster "", domain ""
   Loading event handler: Warewulf::Event::Bootstrap
   Loading event handler: Warewulf::Event::DefaultNode
   Loading event handler: Warewulf::Event::DefaultProvisionNode
   Loading event handler: Warewulf::Event::Dhcp
   Loading event handler: Warewulf::Event::DynamicHosts
   Loading event handler: Warewulf::Event::NewObject
   Loading event handler: Warewulf::Event::ProvisionFileDelete
   Loading event handler: Warewulf::Event::Pxe
   Loading event handler: Warewulf::Event::UniqueNode
   Building default configuration for new node(s)
   Building default configuration for new provision node(s)
   Looking for duplicate node(s)
   Building default configuration for new object(s)
   Writing DHCP configuration
   Building iPXE configuration for: n04/18:66:da:68:3f:d0
   Added to data store:  n04:  172.16.0.4/255.255.255.0/18:66:da:68:3f:d0
   ```

   If this doesn't come up, your node may not be set to PXE boot. You'll have to connect the KVM to fix this (by hitting <F12> on boot).
6. Repeat with the rest of the nodes

Once you're done, you should have all of your nodes booted.

Note: you *won't* have BMC access for nodes 04 - 10. We'll use one of our tools to get those next.

HPC Tools

Step 1: Working with **pdsh**

We're going to explore using the pdsh command (commonly pronounce "p-dish"). pdsh will give us an easy way to run commands on more than one node at once. It does this by sshing to the nodes, possibly in parallel and running a specified command. This is the default behavior, though in a moment we will learn another way it can be used.

Let's first look at the usage info for pdsh:

```
[root@master ~]# pdsh -h
Usage: pdsh [-options] command ...
-S                  return largest of remote command return values
-h                  output usage menu and quit
-V                  output version information and quit
-q                  list the option settings and quit
-b                  disable ^C status feature (batch mode)
-d                  enable extra debug information from ^C status
-l user             execute remote commands as user
-t seconds          set connect timeout (default is 10 sec)
-u seconds          set command timeout (no default)
-f n                use fanout of n nodes
-w host,host,...    set target node list on command line
-x host,host,...    set node exclusion list on command line
-R name             set rcmd module to name
-M name,...         select one or more misc modules to initialize first
-N                  disable hostname: labels on output lines
-L                  list info on all loaded modules and exit
available rcmd modules: ssh,exec (default: ssh)
```

We see that the −w option can be used to specify a list of hosts to run on. The −f option can be used to control the "fanout" of the command, i.e. how many nodes it runs on in parallel. Let's start simple:

```
[root@master ~]# pdsh -f1 -w n01,n02 hostname
n01: n01
n02: n02
```

This ran hostname on nodes n01 and n02. By default, pdsh puts "<host>:" in front of the output for each host it runs on.

Specifying a comma-separated list of hosts could get very long, fortunately, pdsh supports range specifications:

```
[root@master ~]# pdsh -w n[01-03] hostname
n01: n01
n03: n03
n02: n02
```

Note that we let it run in parallel, so our answers came out of order (by default −f is 32).

This syntax is probably fine for our small cluster, but let's take it a step further. pdsh supports add-on modules. We're going to use a module that applies gender specifications through libgender to our nodes. To do this, we need an additional package:

```
[root@master ~]# yum -y install pdsh-mod-genders-ohpc
```

Now edit the file /etc/genders and put the following in it:

```
n01 first
n10 last
n[01-03] three
n07 lucky=true,seven
n[01-06],n[08-10] lucky=false
n[01-10] compute
```

The format of this file is:

```
<nodespec> attr[=value],...
```

If we look at pdsh –h we see we have new options. Let's try a couple of these:

```
[root@master ~]# pdsh -A hostname
n02: n02
n01: n01
n03: n03
... (the rest of the nodes)
```

We should see that we get a response from all of our nodes.

We can also specify based on attributes and attribute/value pairs:

```
[root@master ~]# pdsh -g first hostname
n01: n01
```

```
[root@master ~]# pdsh -g lucky=true hostname
n07: n07
```

```
[root@master ~]# pdsh -f1 -g lucky=false hostname
n01: n01
n02: n02
n03: n03
n04: n04
n05: n05
n06: n06
n08: n08
n09: n09
n10: n10
```

```
[root@master ~]# pdsh -f1 -A -x n07 hostname
n01: n01
n02: n02
n03: n03
n04: n04
n05: n05
n06: n06
```

```
n08: n08
n09: n09
n10: n10
```

Both of these last commands exclude n07.

Note that you may want –f1 if you want to see a consistent, sequential list. It will run much slower though.

Setting up the rest of the BMCs

pdsh is pretty handy. Let's use it for something useful. Let's get the remaining BMC MAC addresses using pdsh to run ipmitool locally on the nodes (we made sure we installed it in the image earlier):

```
[root@master ~]# pdsh -f1 -w n[04-10] ipmitool lan print
n04: Set in Progress          : Set Complete
n04: Auth Type Support        : MD5
n04: Auth Type Enable         : Callback : MD5
n04:                          : User     : MD5
n04:                          : Operator : MD5
n04:                          : Admin    : MD5
n04:                          : OEM      :
n04: IP Address Source        : DHCP Address
n04: IP Address               : 172.16.0.101
n04: Subnet Mask              : 255.255.255.0
n04: MAC Address              : 18:66:da:68:3e:40
...(lots and lots of output)
```

We got our MAC addresses, but we got way too much output. Let's write a command to filter for just what we want.

```
[root@master ~]# pdsh -f1 -w n[04-10] ipmitool lan print | grep "MAC
↪  Address"
n04: MAC Address              : 18:66:da:68:3e:40
n05: MAC Address              : 18:66:da:68:4b:14
n06: MAC Address              : 18:66:da:68:41:3a
...
```

Better, but let's grab *just* the MAC address, and let's store it in a file:

```
[root@master ~]# pdsh -f1 -w n[04-10] ipmitool lan print | grep "MAC
↪  Address" | awk '{print $NF}' | tee /tmp/bmc-macs.txt
18:66:da:68:3e:40
18:66:da:68:4b:14
18:66:da:68:41:3a
...
```

We used awk to get the last space separated column (NF is a special variable that means "number of columns", $NF then means "value of the last column"). tee is a special command that both prints the output and outputs it to a file.

Here's an awk line that will generate the "host" lines we need for our dhcpd.conf:

```
[root@master ~]# awk '{printf "host n%02d-bmc { hardware ethernet %s;
↪  fixed-address 172.16.0.1%02d; }\n", NR+3, $1, NR+3}' /tmp/bmc-macs.txt
...
host n08-bmc { hardware ethernet 18:66:da:68:4b:14; fixed-address
↪  172.16.0.108; }
host n09-bmc { hardware ethernet 18:66:da:68:41:3a; fixed-address
↪  172.16.0.109; }
host n10-bmc { hardware ethernet 18:66:da:68:41:3a; fixed-address
↪  172.16.0.110; }
```

Copy/paste after the other entries in /etc/warewulf/dhcpd−template.conf.

We now need to regenerate the dhcpd.conf and restart dhcpd:

```
[root@master ~]# wwsh dhcp update
Rebuilding the DHCP configuration
Done.
```

```
[root@master ~]# systemctl restart dhcpd
```

We should see that our BMC get their IP address now. Let's force the issue by running the following commands on all of the BMCs:

```
[root@master ~]# pdsh -A ipmitool user set name 2 admin
[root@master ~]# pdsh -A ipmitool user set password 2 admin
[root@master ~]# pdsh -A ipmitool lan set 1 ipsrc dhcp
[root@master ~]# pdsh -A ipmitool chassis bootdev pxe options=persistent
[root@master ~]# pdsh -A ipmitool mc reset cold
```

These will:

1. make sure the username is "admin"
2. set the password to "admin"
3. set the BMC to use DHCP
4. make PXE the persistent first boot option
5. restart the BMC (won't restart the node itself)

Step 2: Powerman & Conman

Powerman

ipmitool is a powerful command, but it can get tedious entering all of the options and remembering all of the commands.

Let's set up a handy tool called, powerman, that we can use to control and monitor system power through IPMI in a more convenient way.

First, we need to install it:

```
[root@master ~]# yum -y install powerman
```

We are going to want to refer to the BMC by hostname, so let's add some entries to /etc/hosts. In /etc/hosts, before the line:

```
### ALL ENTRIES BELOW THIS LINE WILL BE OVERWRITTEN BY WAREWULF ###
```

Add:

```
# cluster_compute_nodes.bmc
172.16.0.101 bmc-n01 bmc-n01.localdomain
172.16.0.102 bmc-n02 bmc-n02.localdomain
172.16.0.103 bmc-n03 bmc-n03.localdomain
172.16.0.104 bmc-n04 bmc-n04.localdomain
172.16.0.105 bmc-n05 bmc-n05.localdomain
172.16.0.106 bmc-n06 bmc-n06.localdomain
172.16.0.107 bmc-n07 bmc-n07.localdomain
172.16.0.108 bmc-n08 bmc-n08.localdomain
172.16.0.109 bmc-n09 bmc-n09.localdomain
172.16.0.110 bmc-n10 bmc-n10.localdomain
```

The configuration for powerman lives under /etc/powerman:

```
[root@master ~]# ls /etc/powerman
apc7900.dev appro-gb2.dev cyclades-pm10.dev hpmpblade.dev ipmipower.dev
 ↪  rancid-cisco-poe.dev
apc7900v3.dev appro-greenblade.dev cyclades-pm20.dev hpmpcell.dev
 ↪  ipmipower-serial.dev raritan-px4316.dev
apc7920.dev bashfun.dev cyclades-pm42.dev hpmp.dev kvm.dev
 ↪  raritan-px5523.dev
apc8941.dev baytech.dev cyclades-pm8.dev hpmpdome.dev kvm-ssh.dev
 ↪  sentry_cdu.dev
apc.dev baytech-rpc18d-nc.dev dli4.dev ibmbladecenter.dev lom.dev swpdu.dev
apcnew.dev baytech-rpc22.dev dli.dev icebox3.dev openbmc.dev vpc.dev
apcold.dev baytech-rpc28-nc.dev eaton-epdu-blue-switched.dev icebox.dev
 ↪  phantom.dev wti.dev
apcpdu3.dev baytech-rpc3-nc.dev eaton-revelation-snmp.dev ics8064.dev
 ↪  plmpower.dev wti-rps10.dev
apcpdu.dev baytech-snmp.dev hp3488.dev ilom.dev powerman.conf.example
apc-snmp.dev cb-7050.dev hpilo.dev ipmi.dev powerman.dev
```

There are a lot of files here that are used to define different types of devices we know how to interact with. The only one we'll be using is ipmipower. The only file we need to modify is /etc/powerman/powerman.conf. Edit /etc/powerman/powerman.conf, and set its contents to:

```
include "/etc/powerman/ipmipower.dev"

device "ipmi0" "ipmipower" "/usr/sbin/ipmipower -D lanplus -u admin -p
 ↪  admin -h bmc-n[01-10] |&"

node "n[01-10]" "ipmi0" "bmc-n[01-10]"
```

Powerman relies on a special service to operate. This service needs a special directory setup that doesn't get setup automatically:

```
[root@master ~]# mkdir -p /var/run/powerman
[root@master ~]# chown daemon:daemon /var/run/powerman
```

Now enable and start the service:

```
[root@master ~]# systemctl enable powerman
[root@master ~]# systemctl start powerman
```

Check its status:

```
[root@master ~]# systemctl status powerman
 powerman.service - PowerMan
   Loaded: loaded (/usr/lib/systemd/system/powerman.service; disabled;
 ↪  vendor preset: disabled)
   Active: active (running) since Sat 2019-06-08 16:22:21 MDT; 4s ago
  Process: 32375 ExecStart=/usr/sbin/powermand (code=exited,
 ↪  status=0/SUCCESS)
 Main PID: 32377 (powermand)
   CGroup: /system.slice/powerman.service
           32377 /usr/sbin/powermand

Jun 08 16:22:21 master systemd[1]: Started PowerMan.
...
```

The command for using powerman is powerman, but we can use the shorter alias pm. To query the status of power on our nodes:

```
[root@master ~]# pm -q
on:  n[01-10]
off:
unknown
```

We should see that all 10 nodes are on.

To power cycle we use:

```
[root@master ~]# pm -c n01
```

To power off:

```
[root@master ~]# pm -0 n01
```

To power on:

```
[root@master ~]# pm -1 n01
```

Conman

Conman is a similar utility to powerman that can use IPMI SOL (and other devices) to provide remote console access. First, we need to install conman:

```
[root@master ~]# yum -y install conman-ohpc
```

(it may already be installed)

The configuration for conman is in /etc/conman.conf. Set its contents to:

```
SERVER keepalive=ON
SERVER logdir="/var/log/conman"
SERVER logfile="/var/log/conman.log"
SERVER loopback=ON
SERVER pidfile="/var/run/conman.pid"
SERVER resetcmd="/usr/bin/powerman -0 %N; sleep 5; /usr/bin/powerman -1 %N"
SERVER tcpwrappers=ON
#SERVER timestamp=1h
GLOBAL seropts="115200,8n1"
GLOBAL log="/var/log/conman/console.%N"
GLOBAL logopts="sanitize,timestamp"

CONSOLE name="n01" dev="ipmi:bmc-n01"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n02" dev="ipmi:bmc-n02"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n03" dev="ipmi:bmc-n03"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n04" dev="ipmi:bmc-n04"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n05" dev="ipmi:bmc-n05"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n06" dev="ipmi:bmc-n06"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n07" dev="ipmi:bmc-n07"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n08" dev="ipmi:bmc-n08"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
CONSOLE name="n09" dev="ipmi:bmc-n09"
↪ ipmiopts="U:admin,P:admin,W:solpayloadsize"
```

```
CONSOLE name="n10" dev="ipmi:bmc-n10"
 ↪  ipmiopts="U:admin,P:admin,W:solpayloadsize"
```

Now we need to enable and start the conman service:

```
[root@master ~]# systemctl enable conman
Created symlink from
 ↪  /etc/systemd/system/multi-user.target.wants/conman.service to
 ↪  /usr/lib/systemd/system/conman.service.
[root@master ~]# systemctl start conman
```

We should now be able to get a serial console on one of our nodes. Note, you may need to hit <enter> to get the login prompt after connecting.

```
[root@master ~]# conman n01

<ConMan> Connection to console [n01] opened.

CentOS Linux 7 (Core)
Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64

n01 login:
```

To escape conman, you need to hit the escape sequence: & − . (i.e. ampersand-period).

Conman stores console logs under /var/log/conman/console.<hostname>. Try:

```
[root@master ~]# cat /var/log/conman/console.n01
<ConMan> Console [n01] log opened at 2019-06-08 16:50:20 MDT.

<ConMan> Console [n01] connected to <bmc-n01>.

<ConMan> Console [n01] joined by <root@localhost> on pts/0 at 06-08 16:50.
2019-06-08 16:50:40
2019-06-08 16:50:40 CentOS Linux 7 (Core)
2019-06-08 16:50:40 Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64
2019-06-08 16:50:40
2019-06-08 16:50:40 n01 login:
2019-06-08 16:50:40 CentOS Linux 7 (Core)
2019-06-08 16:50:40 Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64
2019-06-08 16:50:40
2019-06-08 16:50:40 n01 login:
<ConMan> Console [n01] departed by <root@localhost> on pts/0 at 06-08 16:50.
```

This can be a very useful feature since it will keep a log of, e.g. the boot console messages of each node.

Step 3: Setting up Infiniband

At this point, all of the hardware on our cluster is functional with the exception of our Infiniband fabric (through much of this guide we will refer to Infiniband as IB). There are six steps to getting the Infiniband fabric fully functional.

1. Install necessary software & drivers on the master
2. Enable/Start services on the master
3. Install necessary software & drivers in the BOS image
4. Re-build the VNFS & setup warewulf IB
5. Re-start the nodes & verify IB

We will go through each of these steps, while also learning some of the commands that help us diagnose the IB fabric.

Install necessary software & drivers on the master

We need to install a suite of software known as Mellanox OFED (OpenFabrics Enterprise Distribution). This is the suite of drivers and tools provided by the vendor that will allow us to enable and use the Mellanox IB cards/switch we have in our clusters.

We should already have a repository setup that contains the OFED software.

```
[root@master ~]# yum repolist
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
repo id      repo name       status
...
mlnx         Mellanox OFED   127
...
repolist: 26,044
```

We can see what it provides:

```
[root@master ~]# yum --disablerepo='*' --enablerepo=mlnx list available
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Available Packages
ar_mgr.x86_64               1.0-0.42.g750eb1e.46101              mlnx
cc_mgr.x86_64               1.0-0.41.g750eb1e.46101
 ↪                                 mlnx
dapl.x86_64                 2.1.10mlnx-OFED.3.4.2.1.0.46101     mlnx
dapl-devel.x86_64           2.1.10mlnx-OFED.3.4.2.1.0.46101     mlnx
dapl-devel-static.x86_64    2.1.10mlnx-OFED.3.4.2.1.0.46101     mlnx
...
```

Of particular interest to us are several packages named mlnx−ofed−∗. These packages are "meta-packages" that will install the software needed for particular tasks. Of particular interest to us is mlnx−ofed−hpc:

```
[root@master ~]# yum info mlnx-ofed-hpc
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Available Packages
Name        : mlnx-ofed-hpc
Arch        : noarch
Version     : 4.6
Release     : 1.0.1.1.rhel7.6
Size        : 6.1 k
Repo        : mlnx
Summary     : MLNX_OFED hpc installer package (with KMP support)
URL         : http://mellanox.com
License     : GPLv2 or BSD
Description : MLNX_OFED hpc installer package (with KMP support)
```

Install this on the master. It will take a little while. Let's use it with the time command to see how long:

```
[root@master ~]# time yum -y install mlnx-ofed-hpc
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Resolving Dependencies
--> Running transaction check
...(lots of package installs)

Complete!

real    5m35.542s
user    5m7.848s
sys     2m53.027s
```

Running yum inside of the time command allowed us to track the time spent on the command. This can be pretty handy. We can see here that the command ran for 5 minutes 35 seconds.

We now have quite a few commands available on our master for dealing with IB. By typing ib<tab><tab> we'll get a list of commands starting with "ib". Most of these are for working with the Infiniband.

```
[root@master ~]# ib<tab><tab>
ib2ib_setup           ibdiagpath            ib_send_lat
ibacm                 ibdiscover.pl         ibstat
ib_acme               ibdmchk               ibstatus
ibaddr                ibdmtr                ibswitches
ib_atomic_bw          ibfindnodesusing.pl   ibswportwatch.pl
ib_atomic_lat         ibgenperm             ibsysstat
```

```
ibcacheedit            ibhosts                ibtopodiff
ibccconfig             ibidsverify.pl         ibtracert
ibccquery              iblinkinfo             ibv_asyncwatch
ibcheckerrors          iblinkinfo.pl          ibv_cc_pingpong
ibcheckerrs            ibmirror               ibv_dcini
ibchecknet             ibnetdiscover          ibv_dctgt
ibchecknode            ibnetsplit             ibv_devices
ibcheckport            ibnlparse              ibv_devinfo
ibcheckportstate       ibnodes                ibv_intf
ibcheckportwidth       ibping                 ibv_polldcinfo
ibcheckstate           ibportstate            ibv_rc_pingpong
ibcheckwidth           ibprintca.pl           ibv_srq_pingpong
ibclearcounters        ibprintrt.pl           ibv_task_pingpong
ibclearerrors          ibprintswitch.pl       ibv_uc_pingpong
ibcongest              ibqueryerrors          ibv_ud_pingpong
ibdatacounters         ibqueryerrors.pl       ibv_umr
ibdatacounts           ib_read_bw             ibv_xsrq_pingpong
ibdev2netdev           ib_read_lat            ib_write_bw
ibdiagm.sh             ibroute                ib_write_lat
ibdiagnet              ibrouters
ibdiagnet_csv2xml.py   ib_send_bw
```

Fortunately, we'll only need a few of these for normal operation.

At the moment, these will generally fail because we don't have the IB interface configured. One of the handy ones is iblinkstat, that check our IB port's link status. Let's try it.

```
[root@master ~]# ibstat
```

This will likely result in an error because the appropriate drivers aren't loaded yet.

Enable/Start services on the master

There are two services we'll need on the master:

1. openibd - this initializes the Infiniband hardware (we'll need this one on the nodes too), including loading the necessary kernel modules. To see what modules it loads, let's first list our modules:

   ```
   [root@master ~]# lsmod
   Module                  Size  Used by
   udp_diag               12801  0
   inet_diag              18949  1 udp_diag
   nfsd                  351388  11
   nfs_acl                12837  1 nfsd
   ... (lots more modules)
   ```

Now enable and start the service:

```
[root@master ~]# systemctl enable openibd
[root@master ~]# systemctl start openibd
```

Run lsmod again:

```
[root@master ~]# lsmod
Module                  Size  Used by
rdma_ucm               26930  0
ib_ucm                 22602  0
rdma_cm                60234  1 rdma_ucm
iw_cm                  43514  1 rdma_cm
ib_ipoib              176895  0
ib_cm                  53141  3 rdma_cm,ib_ucm,ib_ipoib
ib_umad                22093  0
mlx5_fpga_tools        14392  0
mlx5_ib               345327  0
mlx5_core             984868  2 mlx5_ib,mlx5_fpga_tools
mlxfw                  18227  1 mlx5_core
mlx4_en               146420  0
ptp                    19231  2 mlx4_en,mlx5_core
pps_core               19057  1 ptp
mlx4_ib               212206  0
ib_uverbs             127130  4 mlx4_ib,mlx5_ib,ib_ucm,rdma_ucm
ib_core               300520  10 rdma_cm,ib_cm,iw_cm,mlx4_ib,mlx5_ib,i ⌋
 ↪  b_ucm,ib_umad,ib_uverbs,rdma_ucm,ib_ipoib
mlx4_core             360639  2 mlx4_en,mlx4_ib
mlx_compat             29590  15 rdma_cm,ib_cm,iw_cm,mlx4_en,mlx4_ib,m ⌋
 ↪  lx5_ib,mlx5_fpga_tools,ib_ucm,ib_core,ib_umad,ib_uver
devlink                48345  4 mlx4_en,mlx4_ib,mlx4_core,mlx5_core
```

lsmod lists modules in reverse order to when they are loaded (newest on top). These are all modules that the openib service loads.

If we look at our network interfaces, we'll also see that we now have an ib0 interface.

```
[root@master ~]# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
 ↪  mode DEFAULT group default qlen 1000 link/loopback
 ↪  00:00:00:00:00:00 brd 00:00:00:00:00:00
...(the em* interfaces)
5: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 2044 qdisc mq state UP
 ↪  mode DEFAULT group default qlen 256 link/infiniband
 ↪  20:00:08:a6:fe:80:00:00:00:00:00:00:11:22:33:44:77:66:77:12 brd
 ↪  00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:
```

Now that the kernel modules are loaded, let's try our ibstat command again:

```
[root@master ~]# ibstat
CA 'mlx5_0'
    CA type: MT4116
    Number of ports: 1
    Firmware version: 12.25.1020
    Hardware version: 0
    Node GUID: 0x1122334477667711
    System image GUID: 0x248a07030029cc48
    Port 1:
        State: Active
        Physical state: LinkUp
        Rate: 100
        Base lid: 12
        LMC: 0
        SM lid: 11
        Capability mask: 0x2651ec48
        Port GUID: 0x1122334477667712
        Link layer: InfiniBand
```

Infiniband does not natively use the IP protocol, but there is a commonly used emulation layer called IPoIB (IP-over-IB) that allows you to use the IB network for normal IP traffic. This is important for some applications to work. It is often used for applications that use IB to bring up initial IB connections, for instance.

Let's configure the IP address for our ib0 interface. Create the file /etc/sysconfig/network−scripts/ifcfg
with the following contents:

```
TYPE=Infiniband
PROXY_METHOD=none
BROWSER_ONLY=no
BOOTPROTO=static
DEFROUTE=no
IPV4_FAILURE_FATAL=no
IPV6INIT=no
NAME=ib0
DEVICE=ib0
ONBOOT=yes
IPADDR=192.168.0.254
NETMASK=255.255.255.0
```

Now, ifup the interface and verify that it's working:

```
[root@master ~]# ifup ib0
[root@master ~]# ip addr show ib0
5: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 2044 qdisc mq state UP
 ↪  group default qlen 256
```

```
    link/infiniband
 ↪  20:00:08:a6:fe:80:00:00:00:00:00:00:11:22:33:44:77:66:77:12 brd
 ↪  00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:
    inet 192.168.0.254/24 brd 192.168.0.255 scope global noprefixroute
 ↪  ib0
       valid_lft forever preferred_lft forever
    inet6 fe80::1322:3344:7766:7712/64 scope link
       valid_lft forever preferred_lft forever
```

2. opensm - this runs the subnet manager that we will need to resolve routes on our IB fabric. Every IB fabric (or other HSN fabric, generally) needs to have at least one of these on the fabric. Some IB switches support running it on the switch, but we still will generally run it on a server, often the master. Sometimes we'll run more than one for failover, but that requires extra configuration.

   Without the subnet manager, the IB will be unusable. There are a couple of commands we can use to verify the subnet manager. The simplest is sminfo. This just lists what is known about the subnet manager:

   ```
   [root@master ~]# sminfo
   sminfo: iberror: failed: query
   ```

   This is clearly broken.

   Another, more generic diagnostic tool is ibdiagnet. This runs a sequence of diagnostics and will report on their status:

   ```
   [root@master ~]# ibdiagnet
   ...
   Plugin Name                                    Result     Comment
   libibdiagnet_cable_diag_plugin-2.1.1           Succeeded  Plugin loaded
   libibdiagnet_phy_diag_plugin-2.1.1             Succeeded  Plugin loaded

   ---------------------------------------------
   Discovery
   -I- Discovering ... 12 nodes (1 Switches & 11 CA-s) discovered.
   -I- Fabric Discover finished successfully
   ...(lots more output
   ```

   This command has a lot of useful information, and will often be your first tool for diagnosing an IB problem. Errors will start with "−E−", we can grep for these:

   ```
   [root@master ~]# ibdiagnet | grep -E '^-E'
   -E- Subnet Manager Check finished with errors
   -E- Not found master subnet manager in fabric
   -E- FW Check finished with errors
   ```

   This is telling us pretty clearly that we need to have a subnet manager running and we don't. Let's start/enable it:

```
[root@master ~]# systemctl enable opensmd
[root@master ~]# systemctl start opensmd
```

Now let's try again:

```
[root@te-hyperv ~]# sminfo
sminfo: sm lid 11 sm guid 0x248a07030029cc48, activity count 349
 ↪  priority 0 state 3 SMINFO_MASTER
```

If we run ibdiagnet we will also see that the subnet manager errors have gone away. Our fabric is now functional.

Install necessary software & drivers in the BOS image

We have the IB setup for the master. Now we need it in the image. As we saw before it can take a while:

```
[root@master ~]# yum -y --installroot=/opt/ohpc/admin/images/centos7
 ↪  install mlnx-ofed-hpc
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Resolving Dependencies
--> Running transaction check
...

Complete!
```

We only need the openibd service in the image. We can enable it with:

```
[root@master ~]# systemctl --root=/opt/ohpc/admin/images/centos7 enable
 ↪  openibd
```

Re-build the VNFS & setup warewulf IB

We've modified our image, now we'll need to rebuild the VNFS.

```
[root@master ~]# wwvnfs --chroot=/opt/ohpc/admin/images/centos7
 ↪  centos7-mlnx
Creating VNFS image from centos-mlnx
Compiling hybridization link tree                          : 0.17 s
Building file list                                         : 0.54 s
Compiling and compressing VNFS                             : 19.82 s
Adding image to datastore                                  : 46.42 s
Wrote a new configuration file at: /etc/warewulf/vnfs/centos-mlnx.conf
Total elapsed time                                         : 66.96 s
```

Note that we used a new image name, centos7−mlnx, instead of centos7−base that we used before. We'll see why in a moment.

We can see that the VNFS is imported:

```
[root@master ~]# wwsh vnfs list
VNFS NAME            SIZE (M)   ARCH       CHROOT LOCATION
centos7-mlnx         399.6      x86_64     /opt/ohpc/admin/images/centos7
centos7-base         261.8      x86_64     /opt/ohpc/admin/images/centos7
```

We are also going to need our ib0 interface's IPoIB configured on the nodes. We will use warewulf files and network config objects to do this.

OpenHPC provided us with a useful template to get started with this. This template will auto-fill an ifcfg −ib0 interface on each node with the information that we tell warewulf about their ib0 interfaces.

```
[root@master ~]# cat /opt/ohpc/pub/examples/network/centos/ifcfg-ib0.ww
DEVICE=ib0
BOOTPROTO=static
IPADDR=%{NETDEVS::IB0::IPADDR}
NETMASK=%{NETDEVS::IB0::NETMASK}
ONBOOT=yes
NM_CONTROLLED=no
DEVTIMEOUT=5
```

We need to import the file into warewulf's file list and add it to the provision set for each node.

```
[root@master ~]# wwsh file import
 ↪  /opt/ohpc/pub/examples/network/centos/ifcfg-ib0.ww
[root@master ~]# wwsh file list
dynamic_hosts           :  rw-r--r-- 0   root root             1501
 ↪  /etc/hosts
group                   :  rw-r--r-- 1   root root              619
 ↪  /etc/group
ifcfg-ib0.ww            :  rw-r--r-- 1   root root              133
 ↪  /opt/ohpc/pub/examples/network/centos/ifcfg-ib0.ww
passwd                  :  rw-r--r-- 1   root root             1370
 ↪  /etc/passwd
shadow                  :  rw-r----- 1   root root              890
 ↪  /etc/shadow
```

We need to tell warewulf that this file needs to live in a different location on the nodes than it does on the master. We can do this with the wwsh file set command:

```
[root@master ~]# wwsh -y file set ifcfg-ib0.ww
 ↪  --path=/etc/sysconfig/network-scripts/ifcfg-ib0
About to apply 1 action(s) to 1 file(s):
     SET: PATH                 = /etc/sysconfig/network-scripts/ifcfg-ib0

Proceed?
```

This remapped its location to /etc/sysconfig/network−scripts/ifcfg−ib0.

We need to tell warewulf's object store about the ib0 information so this template can get filled. We can do this in one command with a for loop:

```
[root@master ~]# for i in $(seq -w 1 10); do wwsh -y node set n$i -D ib0
 ↪  --ipaddr=192.168.0.$i --netmask=255.255.255.0; done
```

To verify that the changes were made:

```
[root@master ~]# wwsh node print n01
#### n01
 ↪  ###########################################################################
...
            n01: ib0.HWADDR       = UNDEF
            n01: ib0.HWPREFIX     = UNDEF
            n01: ib0.IPADDR       = 192.168.0.1
            n01: ib0.NETMASK      = 255.255.255.0
            n01: ib0.NETWORK      = UNDEF
            n01: ib0.GATEWAY      = UNDEF
            n01: ib0.MTU          = UNDEF
            n01: ib0.FQDN         = UNDEF
...
```

Warewulf now knows about the ib0 IP address and netmask, so it can fill out the template for each node.

Finally, we need to add this file to the file list for all of our nodes:

```
[root@master ~]# wwsh -y provision set n[01-10] --fileadd=ifcfg-ib0.ww
```

To verify:

```
[root@master ~]# wwsh provision print n01
#### n01
 ↪  ###########################################################################
...j
            n01: FILES            =
 ↪  dynamic_hosts,group,ifcfg-ib0.ww,passwd,shadow
...
```

We're ready to actually use these changes.

Re-start the nodes & verify IB

The final step to deploying our changes on the nodes is to reboot them into the new image. This is the general workflow for modifying software/configuration on nodes. Part of the stateless model for clustering is making sure that the master fully specifies the operating system for the node. The way we make sure this is the case is by (generally, sometimes we make exceptions) installing new software by loading a clean operating system image on the node, i.e. rebooting.

Remember that we stored this new VNFS with a new name, centos−mlnx. One reason to do this in a real-world situation is that it allows us to test a change on, e.g. a single node without changing the VNFS for all nodes. If we look at the provision list:

```
[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP             FILES
===========================================================================
↪  =====
DEFAULT             centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n01                 centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n02                 centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n03                 centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
...
```

All of our nodes are still mapped to the centos7−base image. We'll start by remapping just n01:

```
[root@master ~]# wwsh provision set n01 -V centos7-mlnx
Are you sure you want to make the following changes to 1 node(s):

     SET: VNFS                 = centos7-mlnx

Yes/No> Yes

[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP             FILES
===========================================================================
↪  =====
DEFAULT             centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n01                 centos7-mlnx    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n02                 centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
```

```
n03                 centos7-base    3.10.0-957.12.2.el...
 ↪  dynamic_hosts,grou...
...
```

Now let's reboot the node. We could use pm to do a *hard* reboot, but it's safer to log in to the node and tell it to reboot.

```
[root@master ~]# ssh n01 systemctl reboot
Connection to n01 closed by remote host.
```

It's often a good idea, especially when testing a new image, to watch the node boot with conman:

```
[root@master ~]# conman n01

<ConMan> Connection to console [n01] opened.
[73286.941232] Restarting system.
...
Starting the provision handler:
 * adhoc-pre
 ↪   OK
 * ipmiconfig Auto configuration not activated
 ↪  SKIPPED
 * filesystems
 ↪  RUNNING
   * mounting /
 ↪   OK
 * filesystems
 ↪   OK
 * getvnfs
 ↪  RUNNING
   * fetching centos7-mlnx (ID:15)
...
[  OK  ] Started mlnx_interface_mgr - configure ib0.
[  OK  ] Started openibd - configure Mellanox devices.
         Starting LSB: Bring up/down networking...
...
CentOS Linux 7 (Core)
Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64

n01 login:
```

Looks like it booted, let's exist conman (&.) and ssh to the node to check it.

```
[root@master ~]# ssh n01
[root@n01 ~]# ls
yum-ww.conf
[root@n01 ~]# ibstat
```

```
CA 'mlx5_0'
    CA type: MT4115
    Number of ports: 1
    Firmware version: 12.17.2020
    Hardware version: 0
    Node GUID: 0x248a07030029cfdc
    System image GUID: 0x248a07030029cfdc
    Port 1:
        State: Active
        Physical state: LinkUp
        Rate: 100
        Base lid: 1
        LMC: 0
        SM lid: 11
        Capability mask: 0x2651e848
        Port GUID: 0x248a07030029cfdc
        Link layer: InfiniBand
```

Make sure it sees the subnet manager:

```
[root@n01 ~]# sminfo
sminfo: sm lid 11 sm guid 0x248a07030029cc48, activity count 1392 priority
 ↪  0 state 3 SMINFO_MASTER
```

Here's another handy command to see what's going on with the IB fabric (especially after more than one node is up):

```
[root@n01 ~]# iblinkinfo
...(output is too big to print here)
```

iblinkinfo will tell you the status of all of the links on the IB fabric.

Finally, let's make sure our IPoIB got set:

```
[root@n01 ~]# ip addr show ib0
7: ib0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 2044 qdisc pfifo_fast state
 ↪  UP group default qlen 256
    link/infiniband
 ↪  80:00:00:68:fe:80:00:00:00:00:00:00:24:8a:07:03:00:29:cf:dc brd
 ↪  00:ff:ff:ff:ff:12:40:1b:ff:ff:00:00:00:
    inet 192.168.0.1/24 brd 192.168.0.255 scope global ib0
       valid_lft forever preferred_lft forever
    inet6 fe80::268a:703:29:cfdc/64 scope link
       valid_lft forever preferred_lft forever
```

Everything looks good here. Exit out of ssh and get back to the master.

Let's go ahead and transition all of the nodes to the new image and reboot them.

```
[root@master ~]# wwsh provision set n[2-10] -V centos7-mlnx
Are you sure you want to make the following changes to 2 node(s):

     SET: VNFS                 = centos7-mlnx

Yes/No> Yes
```

```
[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP             FILES
============================================================================
↪  =====
DEFAULT             centos7-base    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n01                 centos7-mlnx    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n02                 centos7-mlnx    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
n03                 centos7-mlnx    3.10.0-957.12.2.el...
↪  dynamic_hosts,grou...
...
```

We can use pdsh to reboot all of the rest of the nodes:

```
[root@master ~]# pdsh -A -x n01 systemctl reboot
n02: Connection to n02 closed by remote host.
n01: Connection to n02 closed by remote host.
n03: Connection to n02 closed by remote host.
...
```

It's a good idea to watch one of these with conman to get a sense of when they come up:

```
[root@master ~]# conman n02

<ConMan> Connection to console [n02] opened.
[73910.939371] Restarting system.
...
CentOS Linux 7 (Core)
Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64

n02 login:
```

Now, verify that all of the nodes are up with pdsh:

```
[root@master ~]# pdsh -A uptime
n03:  12:08:54 up 2 min,  0 users,  load average: 0.14, 0.10, 0.04
n02:  12:08:56 up 2 min,  0 users,  load average: 0.13, 0.10, 0.04
n01:  12:08:54 up 12 min,  0 users,  load average: 0.00, 0.01, 0.04
...
```

Finally, run an `iblinkinfo` and see that all of your IB links in the fabric look correct. Things to look for include:

Every link should report as:

```
4X      25.78125 Gbps Active/  LinkUp
```

- Are all of the expected links there?
- Does everything report LinkUp?
- Does everything report the right speed?

Finally, make sure ibdiagnet doesn't report any errors.

We now have working Infiniband!

Step 4: Deploying Slurm

Up until now, we have no way to actually make our cluster do work than to ssh to a node and launch a process. Most clusters use a "Work Load Manager" (WLM) or "Queuing system" to manage work to be done on a cluster. This serves multiple purposes:

- Provides a nice interface for launch work on a cluster, possibly across multiple nodes at once
- Provides a queue to make sure we only run as much work as we can handle at once
- Generally provides SLA ("Service Level Agreement") policies to make sure different users get the share of the system that they should
- Provides accounting information on who has used what resources

We will now deploy a popular WLM called "Slurm". Our workflow will be similar to what we followed in the Infiniband step:

1. Setup software/services on the master
2. Setup software/services on the nodes
3. Map necessary configuration to the nodes
4. Create a VNFS and test it on one node
5. Remap our image to the rest of the nodes and reboot

Setup software/services on the master

Slurm has multiple pieces that are needed in different locations. The only piece strictly needed on the master is slurmctld. This is the services that control a Slurm cluster. The nodes will run a slurmd service that synchronizes them with the slurmctld. An optional component, slurmdbd, keeps accounting information in a database. We will not be deploying slurmdbd.

Additionally, all Slurm processes require a service called munge to be running, but on the master and compute nodes. munge handles internal authorization for Slurm. It has a special secret key that must be shared across the cluster.

We will only install slurmctld on the master. We will also install the package slurm−example−configs−ohpc so that we have some example configurations.

On the master:

```
[root@master ~]# yum -y install slurm-ohpc slurm-slurmctld-ohpc
 ↪  slurm-example-configs-ohpc
Loaded plugins: fastestmirror
Loading mirror speeds from cached hostfile
Resolving Dependencies
--> Running transaction check
...

Complete!
```

We need to write a Slurm configuration. This is stored at /etc/slurm/slurm.conf. We need to edit this file. It has the form <parameter>=<value>. The following <parameters> need to be modified from the default:

```
ClusterName=<your_cluster_name>
ControlMachine=<your_master_hostname>
SallocDefaultCommand="/usr/bin/srun -n1 -N1 --mem-per-cpu=0 --pty
 ↪  --preserve-env --mpi=none $SHELL"
NodeName=n[01-10] Sockets=2 CoresPerSocket=16 ThreadsPerCore=1 State=UNKNOWN
PartitionName=normal Nodes=n[01-10] Default=YES MaxTime=24:00:00 State=UP
```

Now let’s enable and start services:

```
[root@master ~]# systemctl enable munge slurmctld
[root@master ~]# systemctl start munge slurmctld
```

```
[root@master ~]# systemctl status munge
 munge.service - MUNGE authentication service
   Loaded: loaded (/usr/lib/systemd/system/munge.service; enabled; vendor
 ↪  preset: disabled)
   Active: active (running) since Sun 2019-06-09 12:37:33 MDT; 2min 33s ago
...
```

```
[root@master ~]# systemctl status slurmctld
 slurmctld.service - Slurm controller daemon
   Loaded: loaded (/usr/lib/systemd/system/slurmctld.service; enabled;
 ↪  vendor preset: disabled)
   Active: active (running) since Sun 2019-06-09 12:37:40 MDT; 2min 46s ago
...
```

We’ll see that munge created a munge key file:

```
[root@master ~]# ls -l /etc/munge/munge.key
-r--------. 1 munge munge 1024 Jun  9 12:28 /etc/munge/munge.key
```

We will have to share this key with our nodes. It's extremely important that the permissions on the key are correct, as seen here.

An extremely common reason for Slurm to be broken is that the munge service is broken. The most common reason for the munge service to be broken is that there is something wrong with the munge key.

Finally, if all has gone well, we can run some Slurm commands to test:

```
[root@master ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 1-00:00:00      0    n/a
```

sinfo is a general status command for Slurm. It can be used to tell you the state of your cluster. Everything is down right now because Slurm isn't running on our nodes yet.

When we need more detail, we can use the scontrol command. It has many options and functions. Here we can see the status of our "partition":

```
[root@master ~]# scontrol show partition
PartitionName=normal
   AllowGroups=ALL AllowAccounts=ALL AllowQos=ALL
   AllocNodes=ALL Default=YES QoS=N/A
   DefaultTime=NONE DisableRootJobs=NO ExclusiveUser=NO GraceTime=0
 ↪  Hidden=NO
   MaxNodes=UNLIMITED MaxTime=1-00:00:00 MinNodes=0 LLN=NO
 ↪  MaxCPUsPerNode=UNLIMITED
   Nodes=n[1-10]
   PriorityJobFactor=1 PriorityTier=1 RootOnly=NO ReqResv=NO
 ↪  OverSubscribe=NO
   OverTimeLimit=NONE PreemptMode=OFF
   State=UP TotalCPUs=320 TotalNodes=10 SelectTypeParameters=NONE
   JobDefaults=(null)
   DefMemPerNode=UNLIMITED MaxMemPerNode=UNLIMITED
```

A "partition" is a logical segment of a slurm cluster. It contains a sub-grouping of nodes. Ours only has one called "normal", but you can have many different partitions. You may be able to spot the line in slurm.conf that defines this partition.

Setup software/services on the nodes

We need slurmd on our nodes, so we'll need to install into the BOS:

```
[root@master ~]# yum -y --installroot=/opt/ohpc/admin/images/centos7
 ↪  install slurm-ohpc slurm-slurmd-ohpc slurm-example-configs-ohpc
```

We need our munge.key setup in the image. We could use wwsh file for this, but we don't expect the munge.key to ever change, so we can build it directly into our image:

```
[root@master ~]# cp -av /etc/munge/munge.key
 ↪  /opt/ohpc/admin/images/centos7/etc/munge/
cp: overwrite ‘/opt/ohpc/admin/images/centos7/etc/munge/munge.key’? y
‘/etc/munge/munge.key’ ->
 ↪  ‘/opt/ohpc/admin/images/centos7/etc/munge/munge.key’

[root@master ~]# ls -l /opt/ohpc/admin/images/centos7/etc/munge/
total 4
-r--------. 1 munge munge 1024 Jun  9 12:28 munge.key

[root@master ~]# md5sum /etc/munge/munge.key
 ↪  /opt/ohpc/admin/images/centos7/etc/munge/munge.key
a2a35e99d51f6c686b31046724196c72  /etc/munge/munge.key
a2a35e99d51f6c686b31046724196c72
 ↪  /opt/ohpc/admin/images/centos7/etc/munge/munge.key
[root@master ~]#
```

We’ve verified that we have identical munge.key files in the image and master, and that the permissions are correct.

Now we need to enable the munge and slurmd services in the BOS:

```
[root@master ~]# systemctl --root=/opt/ohpc/admin/images/centos7 enable
 ↪  munge slurmd
Created symlink /opt/ohpc/admin/images/centos7/etc/systemd/system/multi-use⌋
 ↪  r.target.wants/slurmd.service, pointing to
 ↪  /usr/
```

Map necessary configuration to the nodes

We will map three configuration files into the node:

- /etc/slurm/slurm.conf - the main slurm configuration
- /etc/slurm/slurm.epilog.clean - a script that will run after each job finishes (we won’t change it from the defaultj)
- /etc/slurm/cgroup.conf - a configuration file that slurm uses to setup cgroups. We won’t need to modify this, but it must be in place for slurmd.

First, we need to create /etc/slurm/cgroup.conf. We’ll just copy it from the example:

```
[root@master ~]# cp /etc/slurm/cgroup.conf.example /etc/slurm/cgroup.conf
```

Now, let’s add these three files to warewulf. We’ll use the wwsh shell this time:

```
[root@master ~]# wwsh
Warewulf> file import /etc/slurm/slurm.conf
Warewulf> file import /etc/slurm/cgroup.conf
Warewulf> file import /etc/slurm/slurm.epilog.clean
Warewulf> file list
```

```
cgroup.conf             :  rw-r--r-- 1   root root              216
 ↪  /etc/slurm/cgroup.conf
dynamic_hosts           :  rw-r--r-- 0   root root             1637
 ↪  /etc/hosts
group                   :  rw-r--r-- 1   root root              619
 ↪  /etc/group
ifcfg-ib0.ww            :  rw-r--r-- 1   root root              133
 ↪  /etc/sysconfig/network-scripts/ifcfg-ib0
passwd                  :  rw-r--r-- 1   root root             1370
 ↪  /etc/passwd
shadow                  :  rw-r----- 1   root root              890
 ↪  /etc/shadow
slurm.conf              :  rw-r--r-- 1   root root             2253
 ↪  /etc/slurm/slurm.conf
slurm.epilog.clean      :  rwxr-xr-x 1   root root              888
 ↪  /etc/slurm/slurm.epilog.clean
```

Now, we need to map those files to our nodes:

```
[root@master ~]# wwsh -y provision set n[01-10]
 ↪  --fileadd=slurm.conf,slurm.epilog.clean,cgroup.conf
Are you sure you want to make the following changes to 3 node(s):

     ADD: FILES                = slurm.conf,slurm.epilog.clean,cgroup.conf

Yes/No>
[root@master ~]# wwsh provision print n01
#### n01
 ↪  #########################################################################
...
            n01: FILES            = cgroup.conf,dynamic_hosts,group,ifcfg-i
 ↪  b0.ww,passwd,shadow,slurm.conf,slurm.epilog.clea
...
```

Our files are now mapped in.

The slurm and munge services each added new users to our cluster, so we need to resync our passwd, group, and shadow files, otherwise the special users won't exist for our nodes:

```
[root@master ~]# wwsh file resync
```

We can verify we have the right version of the passwd file:

```
[root@master ~]# wwsh file print passwd
#### passwd
 ↪  ######################################################################
passwd          : ID                 = 1
passwd          : NAME               = passwd
passwd          : PATH              = /etc/passwd
passwd          : ORIGIN            = /etc/passwd
passwd          : FORMAT            = data
passwd          : CHECKSUM          = 2861405b284f1f4bc7979873bffcff54
passwd          : INTERPRETER       = UNDEF
passwd          : SIZE              = 1500
passwd          : MODE              = 0644
passwd          : UID               = 0
passwd          : GID               = 0
[root@master ~]# md5sum /etc/passwd
2861405b284f1f4bc7979873bffcff54  /etc/passwd
```

The checksum is correct.

Create a VNFS and test it on one node

Once again, we'll create a VNFS with a new name:

```
[root@master ~]# wwvnfs --chroot=/opt/ohpc/admin/images/centos7
↪  centos7-slurm
Creating VNFS image from centos7-slurm
Compiling hybridization link tree                           : 0.18 s
Building file list                                          : 0.50 s
Compiling and compressing VNFS                              : 21.68 s
Adding image to datastore                                   : 49.86 s
Wrote a new configuration file at: /etc/warewulf/vnfs/centos7-slurm.conf
Total elapsed time                                          : 72.23 s
```

```
[root@master ~]# wwsh vnfs list
VNFS NAME            SIZE (M)   ARCH       CHROOT LOCATION
centos7-base         261.8      x86_64     /opt/ohpc/admin/images/centos7
centos7-mlnx         399.6      x86_64     /opt/ohpc/admin/images/centos7
centos7-slurm        426.6      x86_64     /opt/ohpc/admin/images/centos7
```

Now we'll follow the same procedure as before.

```
[root@master ~]# wwsh provision set n01 -V centos7-slurm
Are you sure you want to make the following changes to 1 node(s):

     SET: VNFS                 = centos7-slurm

Yes/No> Yes
```

```
[root@master ~]# ssh n01 systemctl reboot
Connection to n01 closed by remote host.
```

```
[root@master ~]# conman n01
```

```
<ConMan> Connection to console [n01] opened.
[ 4108.855476] Restarting system.
...
[  OK  ] Started MUNGE authentication service.
...
[  OK  ] Started Slurm node daemon.
...
CentOS Linux 7 (Core)
Kernel 3.10.0-957.12.2.el7.x86_64 on an x86_64

n01 login:
```

Now, on the master we can try sinfo again:

```
[root@master ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 1-00:00:00      2   unk* n[02-03]
normal*      up 1-00:00:00      1   idle n01
```

We see that n01 is idle.

Let's try it out:

```
[root@master ~]# salloc
salloc: Granted job allocation 5
[root@n01 ~]# hostname
n01
[root@n01 ~]# squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
                 5    normal       sh     root  R       0:08      1 n01
[root@n01 ~]# exit
```

We'll have the opportunity to use Slurm more later.

Remap our image to the rest of the nodes and reboot

Now that everything is working, let's set the VNFS for the rest of the nodes to centos7−slurm and reboot.

```
[root@master ~]# wwsh provision set n[02-10] -V centos7-slurm
Are you sure you want to make the following changes to 2 node(s):

     SET: VNFS                 = centos7-slurm

Yes/No> Yes
```

```
[root@master ~]# wwsh provision list
NODE                VNFS            BOOTSTRAP             FILES
===========================================================================⌟
 ↪  =====
DEFAULT             centos7-base    3.10.0-957.12.2.el...
 ↪  dynamic_hosts,grou...
n01                 centos7-slurm   3.10.0-957.12.2.el...
 ↪  cgroup.conf,dynami...
n02                 centos7-slurm   3.10.0-957.12.2.el...
 ↪  cgroup.conf,dynami...
n03                 centos7-slurm   3.10.0-957.12.2.el...
 ↪  cgroup.conf,dynami...
...
```

```
[root@master ~]# pdsh -A -x n01 systemctl reboot
n03: Connection to n03 closed by remote host.
pdsh@master: n03: ssh exited with exit code 255
n02: Connection to n02 closed by remote host.
pdsh@master: n02: ssh exited with exit code 255
...
```

```
[root@master ~]# conman n02
...
```

Now let's check sinfo:

```
[root@master ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 1-00:00:00      1   idle n01
normal*      up 1-00:00:00      2   down n[02-10]
```

Why does it show them as down? This is a little misleading. If they were actually down the would say ∗down not down. The ∗ means that they cannot be contacted. Whenever a resource goes away, by default we need to resume it.

```
[root@master ~]# scontrol update nodename=n[02-10] state=resume
```

```
[root@master ~]# sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
normal*      up 1-00:00:00     10   idle n[01-10]
```

That's better. We now have 10 nodes we can run slurm jobs on!

Step 5: Program environments with `lmod`

Users of a compute cluster often need various programming environments (PEs) available for their applications. For instance, one package may require OpenMPI 3.x while another uses

OpenMPI 4.x. Or perhaps one requires GCC 7.x while the other requires the PGI compiler suite. We need a convenient way to manage this type of software for the user.

lmod is a package that takes different software environment specifications and makes them available to the user through a convenient user interface. Typically, the software that lmod manages is going to live on a shared filesystem. In our case, we will install this kind of tool under /opt/ohpc/pub, which the nodes have access to through the NFS share.

OpenHPC provides a number of packages that will automatically be set up to use lmod. We already built base lmod support into our images from the beginning. Let's start by installing some of the development tools OpenHPC provides.

On the master (this will take several minutes):

```
[root@master ~]# yum -y install EasyBuild-ohpc \
                    gnu8-compilers-ohpc \
                    hwloc-ohpc \
                    llvm5-compilers-ohpc \
                    lmod-defaults-gnu8-openmpi3-ohpc \
                    mpich-gnu8-ohpc \
                    ohpc-autotools \
                    ohpc-gnu8-io-libs \
                    ohpc-gnu8-mpich-parallel-libs \
                    ohpc-gnu8-openmpi3-parallel-libs \
                    ohpc-gnu8-perf-tools \
                    ohpc-gnu8-python-libs \
                    ohpc-gnu8-runtimes \
                    ohpc-gnu8-serial-libs \
                    openmpi3-pmix-slurm-gnu8-ohpc \
                    spack-ohpc \
                    valgrind-ohpc
```

```
...(lots of packages install)
```

Now, log out and log back into the master. We can list the available lmod modules:

```
[root@master ~]# module avail
```

```
------------------------- /opt/ohpc/pub/moduledeps/gnu8-openmpi3
↪  -------------------------
   adios/1.13.1     mpiP/3.4.1              pnetcdf/1.11.0      scorep/4.1
   boost/1.69.0     mumps/5.1.2             ptscotch/6.0.6
↪  sionlib/1.7.2
   dimemas/5.3.4    netcdf-cxx/4.3.0        py2-mpi4py/3.0.0    slepc/3.10.2
   extrae/3.5.2     netcdf-fortran/4.4.5    py2-scipy/1.2.1
↪  superlu_dist/6.1.1
   fftw/3.3.8       netcdf/4.6.2            py3-mpi4py/3.0.0    tau/2.28
   hypre/2.15.1     opencoarrays/2.2.0      py3-scipy/1.2.1
 ↪  trilinos/12.12.1
   imb/2018.1       petsc/3.10.3            scalapack/2.0.2
   mfem/3.4         phdf5/1.10.4            scalasca/2.4

------------------------------ /opt/ohpc/pub/moduledeps/gnu8
 ↪  ------------------------------
   R/3.5.2         metis/5.1.0     openblas/0.3.5        py2-numpy/1.15.3
   gsl/2.5         mpich/3.3       openmpi3/3.1.3 (L)    py3-numpy/1.15.3
   hdf5/1.10.4     mvapich2/2.3    pdtoolkit/3.25        scotch/6.0.6
   likwid/4.3.3    ocr/1.0.1       plasma/2.8.0          superlu/5.2.1

------------------------------- /opt/ohpc/admin/modulefiles
 ↪  -------------------------------
   spack/0.12.1

-------------------------------- /opt/ohpc/pub/modulefiles
 ↪  --------------------------------
   EasyBuild/3.8.1           gnu8/8.3.0  (L)    papi/5.6.0
 ↪  valgrind/3.14.0
   autotools          (L)    hwloc/2.0.3        pmix/2.2.2
   charliecloud/0.9.7        llvm5/5.0.1        prun/1.3          (L)
   cmake/3.13.4              ohpc        (L)    singularity/3.1.0

  Where:
   L:  Module is loaded

Use "module spider" to find all possible modules.
Use "module keyword key1 key2 ..." to search for all possible modules
 ↪  matching any of the
"keys".
```

Since these are shared to our nodes, we should be able to get them there as well. Let's start an interactive job to a node and see what we can do. You can do this as your own user:

```
[user@master ~]$ salloc
salloc: Granted job allocation 7
[user@n01 ~]$ module avail
...(should give the same module list)
```

To see what modules you currently have loaded (some modules get loaded by default):

```
[user@n01 ~]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
```

To load a new module:

```
[user@n01 ~]$ R --version
bash: R: command not found
[user@n01 ~]$ module load R
[user@n01 ~]$ R --version
R version 3.5.2 (2018-12-20) -- "Eggshell Igloo"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under the terms of the
GNU General Public License versions 2 or 3.
For more information about these matters see
http://www.gnu.org/licenses/.
```

Note: we could have specified the full R version, e.g. module load R/3.5.2.

We can also unload modules:

```
[user@n01 ~]$ module unload R
[user@n01 ~]$ R --version
bash: R: command not found
```

There are a number of different things you can do with lmod. For instance, you could load several modules you need and “save” that state so you can restore it any time you need it later.

We’ll be using lmod modules later. The specification for lmod modules are written in a language called lua. We will not be going into the particulars of making a module, but you can look at the files in the tree: /opt/ohpc/admin/lmod.

Using Slurm

Overview

In Guide 7.2 we worked through installing and testing the Slurm workload manager. In this guide we’ll get a brief introduction to how to use Slurm from the user perspective.

Anatomy of a Slurm job

A Slurm job consists of an allocation and a number of steps that occur within the allocation. An allocation can be gained directly by use of the salloc command. It can also be obtained through use of a batch script and the sbatch command.

Steps within a job are allocated through the srun command. One allocation may consists of many steps.

An allocation (salloc) is granted a specified amount of resources. For our purposes, we'll stick to three resources: nodes, CPUs (cores), and memory.

A step (srun) is granted a certain number of resources within the allocation. By default, it will be granted all of the resources of the allocation, but it doesn't have to be.

An salloc command has the structure:

```
salloc [options] <command>
```

Some basic options are:

- −N # - number of nodes
- −n # - number of tasks (parallel)
- −c # - number of CPUs per task
- −−mem=# - memory per node
- −−mem−per−cpu=# - memory per CPU
- −−time or −t - limit on runtime

The <command> gets executed locally on the host where the allocation is requested. In this case, its job is primarily to coordinate steps sruns to run on the computing resources.

The allocation is bound to the lifetime of the <command>. When the <command> exits, the allocated resources are returned to the free resource pool.

The format of srun is similar to salloc :

```
srun [options] <command>
```

Many of the same resource selections apply as salloc , but the resources are taken from within the existing allocation.

- −N # - number of nodes
- −n # - number of tasks (parallel)
- −c # - number of CPUs per task
- −−mem=# - memory per node
- −−mem−per−cpu=# - memory per CPU
- −−time or −t - limit on runtime
- −−pty - allocate an interactive session inside the allocation

We can call srun multiple times within one allocation. If our srun does not use all of the resources of our allocation, we could have multiple srun's running at the same time.

If the −−pty otpion is given, the <command> is allocated one of the allocated hosts. In this case, it gives us interactive access to a compute host.

When we run an srun with −N or −n > 1, the process is launched that many times.

Step 1: Simple interactive session

Suppose we want to just poke around on a compute node a bit. We don't need significant resources, and we only need one CPU. This will need to be an interactive session then. We can do this with:

```
[lowell@te-master ~]$ salloc -n1 /bin/bash
salloc: Granted job allocation 9
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
                 9   cluster     bash   lowell  R       0:01      1 te01
[lowell@te-master ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cluster*     up   infinite      1    mix te01
cluster*     up   infinite      8   idle te[02-10]
[lowell@te-master ~]$ srun --pty /bin/bash
[lowell@te01 ~]$ uptime
 20:07:36 up 11:05,  0 users,  load average: 0.00, 0.01, 0.05
[lowell@te01 ~]$ exit
exit
[lowell@te-master ~]$ exit
exit
salloc: Relinquishing job allocation 9
```

In this command we allocated one CPU and our allocation process was bash. We see our running allocation with the squeue command. We see in sinfo command we see that te01 is in the state mix. This means that the node has an allocation running on it, but parts of the node are idle. That happened because we only requested one CPU, so all of the rest are free.

We then ran the srun command with the −−pty option, again running /bin/bash. This gave us an interactive step on the actual node.

Finally, we exited both shells and were told that the allocation was relinquished.

We could have simplified this by making srun the command we run with salloc. In fact this is commonly done:

```
[lowell@te-master ~]$ salloc -n1 srun --pty /bin/bash
salloc: Granted job allocation 19
[lowell@te01 ~]$ exit
exit
salloc: Relinquishing job allocation 19
```

In fact, if we look in our slurm.conf, this is set to be the default action if we give an salloc without a command:

```
[lowell@te-master ~]$ grep SallocDefaultCommand /etc/slurm/slurm.conf
```

```
SallocDefaultCommand="/usr/bin/srun -n1 -N1 --mem-per-cpu=0 --pty
 ↪  --preserve-env --mpi=none $SHELL"
```

So, really, we can just do:

```
[lowell@te-master ~]$ salloc
salloc: Granted job allocation 20
[lowell@te01 ~]$ exit
exit
salloc: Relinquishing job allocation 20
```

Note that this doesn't have to be the default command; we set it up this way.

Step 2: Running a multi-processor application

Suppose we want our application to run on 8 processors, with 1 GB memory per processor. We can make an allocation like this with:

```
[root@te-master ~]# salloc -n8 --mem-per-cpu=1 -N1 /bin/bash
salloc: Granted job allocation 59
[root@te-master ~]# srun hostname
te01
te01
te01
te01
te01
te01
te01
```

We specified both −n8 and −N1 to make sure that we get all 8 tasks on one node.

Notice that srun will, by default, launch one tasks for each allocated task. We can control this with −−cpus−per−task, or −c.

```
[root@te-master ~]# salloc -n1 -N1 -c8 /bin/bash
salloc: Granted job allocation 62
[root@te-master ~]# srun hostname
te01
```

In this example, we got 1 task, but 8 CPUs in the task. This just means that Slurm found a node with 8 free CPUs for our single task.

We could also get 8 tasks on 4 nodes:

```
[root@te-master ~]# srun hostname
te04
te01
te03
te02
```

```
te01
te01
te01
te01
```

Notice that this did not evenly distribute our tasks across our nodes. We need to specifically request that if it is what we want.

```
[root@te-master ~]# salloc -n8 -N4 --tasks-per-node=2 /bin/bash
salloc: Granted job allocation 64
[root@te-master ~]# srun hostname
te01
te01
te04
te03
te02
te04
te02
te03
[root@te-master ~]# exit
exit
salloc: Relinquishing job allocation 64
```

Now there are two tasks on every node.

Using −N, −n, and −c allong with various −−<x>−per−<y> style parameters we can control how our job gets allocated in quite a bit of detail.

Step 3: Batch scripts

Everything we've done so far requires more interaction than we would typically want to do. In reality, we usually want to bundle our jobs up in scripts that can run whenever the resources are available. These are "sbatch" scripts, and can be submitted with sbatch.

An sbatch script is an ordinary shell script, but it has the option of containing lines starting with #SBATCH. These lines can be followed with arbitrary srun/ salloc command arguments. Usually, an sbatch script does all of its work in srun steps just like we have been doing.

Here is an example of a simple sbatch script:

```
#!/bin/bash

#SBATCH --job-name=Test
#SBATCH --ntasks=2
#SBATCH --nodes=1
#SBATCH --cpus-per-task=2
#SBATCH --mem-per-cpu=1
```

```
#SBATCH --output=test_%j.log
pwd; date

srun hostname
srun sleep 60

date
```

This sbatch script asks for a job with 2 tasks, 1 node, 2 cpus per task (a total of 4 CPUs), and 1 GB of memory per task. It is assigned the name "Test". A job name can help us identify jobs in the queue. We have also requested that output be sent to test_<job_id>.log. There are a number of macros like %j for job_id that can be used in these scripts.

While it's not necessary, it's common practice in sbatch scripts to:

- Have a single parameter per #SBATCH line. This makes the file easy to read and easy to edit.
- Use full-length arguments instead of short arguments (e.g. −−ntasks, not −n). This improves readability.

We can submit this sbatch script (assuming it's called test.sbatch):

```
[lowell@te-master ~]$ sbatch test.sbatch
Submitted batch job 68
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
                68   cluster     Test   lowell  R       0:02      1 te01
[lowell@te-master ~]$ sinfo
PARTITION AVAIL  TIMELIMIT  NODES  STATE NODELIST
cluster*     up   infinite      1    mix te01
cluster*     up   infinite      9   idle te[02-10]
[lowell@te-master ~]$ scontrol show job 68
JobId=68 JobName=Test
   UserId=lowell(1000) GroupId=lowell(1000) MCS_label=N/A
   Priority=4294901693 Nice=0 Account=(null) QOS=(null)
   JobState=RUNNING Reason=None Dependency=(null)
   Requeue=1 Restarts=0 BatchFlag=1 Reboot=0 ExitCode=0:0
   RunTime=00:00:09 TimeLimit=UNLIMITED TimeMin=N/A
   SubmitTime=2019-06-14T04:26:35 EligibleTime=2019-06-14T04:26:35
   AccrueTime=Unknown
   StartTime=2019-06-14T04:26:35 EndTime=Unknown Deadline=N/A
   PreemptTime=None SuspendTime=None SecsPreSuspend=0
   LastSchedEval=2019-06-14T04:26:35
   Partition=cluster AllocNode:Sid=te-master:31714
   ReqNodeList=(null) ExcNodeList=(null)
   NodeList=te01
   BatchHost=te01
   NumNodes=1 NumCPUs=4 NumTasks=2 CPUs/Task=2 ReqB:S:C:T=0:0:*:*
   TRES=cpu=4,mem=4M,node=1,billing=4
   Socks/Node=* NtasksPerN:B:S:C=0:0:*:* CoreSpec=*
   MinCPUsNode=2 MinMemoryCPU=1M MinTmpDiskNode=0
   Features=(null) DelayBoot=00:00:00
   OverSubscribe=OK Contiguous=0 Licenses=(null) Network=(null)
   Command=/home/lowell/test.sbatch
   WorkDir=/home/lowell
   StdErr=/home/lowell/test_68.log
   StdIn=/dev/null
   StdOut=/home/lowell/test_68.log
   Power=
```

As you can see, scontrol show job <job−id> will tell you information about a specific job, including its resource limits.

After a minute, the job will complete.

```
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
[lowell@te-master ~]$ cat test_68.log
/home/lowell
Fri Jun 14 04:26:36 MDT 2019
te01
te01
Fri Jun 14 04:27:36 MDT 2019
```

This shows us the output from our job.

We can potentially have a lot of jobs submitted at the same time. The scheduler will launch them as resources become available.

Suppose we run it again, but decide we don't want it anymore. We can scancel the job:

```
[lowell@te-master ~]$ sbatch test.sbatch
Submitted batch job 69
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
                69   cluster     Test   lowell  R       0:02      1 te01
[lowell@te-master ~]$ scancel 69
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
```

The output file for jobs is updated live. We can, for instance, tail −f the output of a running job. We can see this by noting that our canceled job still has an output file, but it is partial:

```
[lowell@te-master ~]$ cat test_69.log
/home/lowell
Fri Jun 14 04:29:07 MDT 2019
te01
te01
srun: Job step aborted: Waiting up to 32 seconds for job step to finish.
slurmstepd: error: *** STEP 69.1 ON te01 CANCELLED AT 2019-06-14T04:29:13
 ↪  ***
slurmstepd: error: *** JOB 69 ON te01 CANCELLED AT 2019-06-14T04:29:13 ***
```

Slurm also tells us that the job was canceled.

Let's modify our job a little to make it have more regular output. Let's call this one test2.sbatch:

```
#!/bin/bash

#SBATCH --job-name=Test2
#SBATCH --ntasks=1
#SBATCH --nodes=1
#SBATCH --cpus-per-task=1
#SBATCH --mem-per-cpu=1
#SBATCH --output=test2_%j.log
pwd; date

srun hostname
srun /bin/bash -c 'for i in {1..60}; do echo $i; sleep 1; done'

date
```

Note that we adjusted our tasks/CPUs to just be one.

We can run this job, and while it's running attach to its output to see what's going on:

```
[lowell@te-master ~]$ sbatch test2.sbatch
Submitted batch job 71
[lowell@te-master ~]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
                71   cluster    Test2   lowell  R       0:02      1 te01
[lowell@te-master ~]$ sattach 71.1
1
2
3
4
5
6
7
```

```
8
9
10
...
```

Note that we must specify <job_id>.<step_id> for sattach, not just the <job_id>.

Conclusion

There are many features for submitting jobs not covered here, but this guide should provide a basic overview of how jobs are submitted in a Slurm cluster. Specifically, we haven't covered how to submit and use MPI jobs, but that will have to be covered later when we actually have an MPI application to run.

Meanwhile, your best resource for submitting slurm jobs are the salloc, srun, and sbatch man pages.

Version Control with Git

Overview

This guide will take you through exercises to gain familiarity with the basics of the git version control system. The guide can be followed on any system which has a reasonably modern version of git is installed; no external servers are required.

Step 1: Getting started with a Git repo

Create and inspect a git repo

We will work out of our $HOME directory throughout this guide. None of this guide should require root access.

To start, we need to create a directory that we will "initialize" as a git repository.

```
$ cd $HOME
$ mkdir git-practice
$ cd git-practice
```

We should now be in the git−practice directory. We can turn this into a git repo with one simple command:

```
$ git init
Initialized empty Git repository in /Users/lowell/git-practice/.git/
```

If we look at the contents of the directory, we will see that we have a directory named .git.

```
$ ls -a
.    ..    .git
```

This .git directory contains the various metadata that will allow git to track changes for files in our git−practice directory. Let's take a look inside of it:

```
$ ls -1F .git
HEAD
config
description
hooks/
info/
objects/
refs/
```

Most of these files/directories are only used for internal tracking for git. config contains the local configuration for the repo. config can be manipulated by hand, but most of its settings can be changed with git commands. hooks contains a set of scripts that will automatically be run when certain actions take place, e.g. a commit. hooks can be a very powerful tool, but we won't be setting them up here.

git status is one of the most common and useful git commands. It's a good idea to run this often to see the current state of our directory.

```
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Setup global config

Before we make any changes to our repository, we'll want to set up some global configuration for our user. Specifically, we'll want to set up our name and our email address to be tied to commits and commit messages. In the following, you should substitute your own information.

```
$ git config --global user.name "J. Lowell Wofford"
$ git config --global user.email lowell@lanl.gov
```

We can view our global configuration:

```
$ git config --global --list
user.name=J. Lowell Wofford
user.email=lowell@lanl.gov
```

There is are a lot of configuration settings available. We can list all of the current settings with:

```
$ git config --list
user.name=J. Lowell Wofford
user.email=lowell@lanl.gov
core.editor=vim
merge.tool=vimdiff
Js-MacBook-Pro:git-clone lowell$ git config --list
credential.helper=osxkeychain
user.name=J. Lowell Wofford
user.email=lowell@lanl.gov
core.editor=vim
merge.tool=vimdiff
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
...
```

Add a file and make your first commit

To start version controlling files, we'll need a file to work with:

```
$ echo "Imma Textfile" > README.md
```

Let's check our status now that we've modified the directory.

```
$ git status
...
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README.md

nothing added to commit but untracked files present (use "git add" to track)
```

It suggests we should add our README.md file. Adding it will move it from being an "untracked" to a "tracked" file. Let's follow the suggestion:

```
$ git add README.md
```

Check the status again:

```
$ git status
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

```
    new file:   README.md
```

We have no added the file. This both set it to tracking, and also staged it for the next commit. Let's go ahead and make our first commit:

```
$ git commit -m "added README"
[master (root-commit) 3519411] added README
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

The −m option specifies a message to associate with the commit. It's a very good idea to give commit messages that are descriptive of what has changed in the directory.

Inspect and make a change

Let's see what our status is now:

```
$ git status
On branch master
nothing to commit, working tree clean
```

We no longer have any changes, staged or unstaged: "working tree clean".

We can look at information about our past commits with git log:

```
$ git log
commit 3519411040534adfd10c059058fe3e715da17dda (HEAD -> master)
Author: J. Lowell Wofford <lowell@lanl.gov>
Date:   Tue Mar 5 08:30:07 2019 -0700

    added README
```

Git log tells us our commit message, who made the change and when it was made. With other options, we can see more and less information. Notice the commit has a hash starting with 351941... This hash is the unique identifier for the commit.

Now, let's modify our README.md.

```
$ echo "...with append" >> README.md
```

Check the status:

```
$ git status
On branch master
Changes not staged for commit:...
    modified:   README.md
no changes added to commit (use "git add" and/or "git commit -a")
```

This is different than our message before. Since README.md is a tracked file, it reports as modified, but *not* staged for commit. We are also offered the git commit −a option now.

We can see how our current state differes from the previous commit:

```
$ git diff
...
--- a/README.md
+++ b/README.md
@@ -1 +1,2 @@
 Imma Textfile
+...with append
```

Let's commit our change to README.md. We can use the −a flag now to auto-add all modified, tracked files (note: it will *not* add untracked files).

```
$ git commit -a -m "update README"
[master 9ed49d5] update README
 1 file changed, 1 insertion(+)
```

git commit −a is handy, but can also be dangerous. You can easily accidentally commit changes you didn't intend. It's a good idea to only use this if you've carefully looked at git status and git diff to know what changes you're about to commit.

We now should have two log messages to commit. Here's a handy, compact version of the log:

```
$ git log --pretty=oneline
4c3a21ef50d10259e9accb2a1c18c1c49ab7c91a (HEAD -> master) update README
b8bbcab814f9f39e852f54911c3470b274a8084b added README
```

This oneline format is especially handy for locating the hash for a known commit. We are going to want to inspect the original commit, so we can grab its hash here. Fortunately, we don't need the whole hash. Git will work if we just pass enough of the hash to uniquely identify it (minimum 5 characters).

Let's checkout our old commit:

```
$ git checkout b8bbc
Note: checking out 'b8bbc'.
You are in 'detached HEAD' state...
```

Our branch should now be in the state of the old commit. The "detached HEAD" warning means that the current "HEAD" (current working directory) does not refer to the latest commit of any known branch, so you can't directly make commits to it (what would it be appending the commit to?).

Double check that we are in the old version:

```
$ cat README.md
Imma Textfile
```

We *do* have the expected old contents of the file. We can now revert back to the latest commit:

```
$ git checkout master
...Switched to branch 'master'
```

Then check the contents again:

```
$ cat README.md
Imma Textfile
...with append
```

We're back where we should be. We used the diff command before to compare a current working directory to the previous commit. We can also compare to any arbitrary commit by hash:

```
$ git diff b8bbc
...@@ -1 +1,2 @@
 Imma Textfile
+...with append
```

Step 2: Working with remotes

We have seen so far how git can work with files in a local directory. In real-world scenarios, we are interested in collaborating through git. Git offers a number of tools for collaboration, but one of the necessary tools is the ability to synchronize similar repos across multiple locations. This is achieved with *remotes*.

Bare repo, remote add & push

We are going to want to make a second directory to work with. We'll call this one git−bare. Let's create it:

```
$ cd $HOME
$ mkdir git-bare
$ cd git-bare
```

We want to git init this repo, but we want a location that does not necessarily have its own working copy (HEAD) but is instead an exchange point for other synchronized working copies. This is called a *bare* repo.

```
$ git init --bare
Initialized empty Git repository in /Users/lowell/dev/git-bare/
```

If we list the contents:

```
$ ls
HEAD  description  info  refs
config  hooks  objects
```

We see that the contents of this *bare* repo look like the contents of .git in a *non-bare* repo. That's because it is not expected to have a working copy, so it just tracks all of the internal data.

We'll go back to our original repo:

```
$ cd ../git-practice
```

Now, we can add our new git−bare repo as a remote for git−practice.

```
$ git remote add origin ../git-bare
```

This means setup the repository at ../ git−bare to be a repository I know how to synchronize with. Give it the name "origin".

Once we have a remote, we can push (synchronize our changes to another remote) our changes to the remote. Because we haven't done this before, our current repo is not tracking anything in "origin". We can both push and setup tracking with:

```
$ git push --set-upstream origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 234 bytes | 234.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To ../git-bare
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

Clone, modify, push

To further explore remotes, we will need a third repo. This time, instead of creating a blank repo and initializing it, we want to start with a new repo that is a complete clone of the existing repo. To do this:

```
$ cd $HOME
$ git clone git-bare git-clone
Cloning into 'git-clone'...
done.
```

This made a copy of git−bare into a new directory, git−clone. Let's inspect it:

```
$ cd $HOME/git-clone
$ cat README.md
Imma Textfile
...with append
```

Now, in our clone, let's create a new file.

```
$ echo "something" > new.txt
```

Let's add it and commit it:

```
$ git add new.txt
$ git commit -m "added something new.txt"
...
```

Checking status:

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Because we got this copy by a clone, we are already tracking the "origin" master. The status tells us that we have changes that "origin" doesn't know about it, and suggests we push them.

Pushing:

```
$ git push
...
To /Users/lowell/dev/git-bare
   f524f51..ac9ad59  master -> master
nothing to commit, working tree clean
```

Note that because we were already tracking, we did not need to provide any arguments to push.

Fetch & pull

Now, let's go back to the repo where we started:

```
$ cd $HOME/git-practice
```

If we look at the status here:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'...
```

We see that nothing appears to have changed. But wait! We know we made a change. What's happening?

Before we can see the change, we need to fetch remote changes. This tells git to download the appropriate metadata from the remote(s) to tell us about them, but not touch our working directory.

```
$ git fetch
...Unpacking objects: 100% (3/3), done.
From ../git-bare
   f524f51..ac9ad59  master     -> origin/master
```

This message lets us know that something has changed with “origin/master”. Let’s get more detail with status:

```
$ git status
...Your branch is behind 'origin/master' by 1 commit, and can be
↪  fast-forwarded.
  (use "git pull" to update your local branch)...
```

Now we see that the “origin” has a commit we *don’t* yet have. We would like to synchronize the changes from the remote to us. This is called a pull:

```
$ git pull
Updating f524f51..ac9ad59
Fast-forward
 new.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new.txt
```

This message tells us that one line was added to a file called new.txt. Let’s take a look:

```
$ cat new.txt
something
```

We got the change! To recap: we made the change in git−clone, we then pushed it to git−bare, and pulled it into git−practice. This is a common workflow. Many collaborators can push their changes to a bare repository and others can pull them down as they want them.

Step 3: Some other useful git commands

There are a couple of commands that may prove helpful as you work with git, and they have been included here.

Reset (soft)

Let’s make sure we’re in our git−practice repo:

```
$ cd $HOME/git-practice
```

Suppose we did the following:

```
$ touch tmp
$ git add .
```

If we look at git status:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    new file:   tmp
```

Wait, we don't want to commit tmp. It's just a temporary file. But, we've now staged it for commit.

Enter git reset. In it's default mode it performs a "soft" reset that just moves files that have been added to the staging area out of the staging area.

```
$ git reset
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    tmp

nothing added to commit but untracked files present (use "git add" to track)
```

We can see that tmp has gone back to being an unstaged, untracked file.

Stash

We're still sitting in git−practice, and we have this untracked tmp file. Suppose someone comes along and asks us to work on something else. We're not ready to commit tmp, but we don't want to lose what we have there.

We can temporarily set aside changes with the stash command:

```
$ git add tmp
$ git stash
Saved working directory and index state WIP on master: ac9ad59 added new.txt
```

If we look in our directory now:

```
$ ls
README.md  new.txt
```

We see that tmp is no longer there.

We can see what stashes we have sitting around (we can have multiples) with:

```
$ git stash list
stash@{0}: WIP on master: ac9ad59 added new.txt
```

We see that we have one stash. It also tells us that it's a Work In Progress (WIP) on a particular commit, ac9ad59. The {0} tells us that it is identified as stash 0.

We can see this particular stash with:

```
$ git stash show 0
 tmp | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
```

Ok, so this stash adds the zero-line file, tmp.

Let's restore our stash:

```
$ git stash pop 0
$ ls
README.md  new.txt  tmp
```

We can see that it has restored our tmp file. Note that the 0 here was not strictly necessary. By default, stash pop will pop the 0th stash, but it is necessary for any other stash.

Conclusion

Git as a lot of features that we won't deal with in this guide. For instance, git's branch/merge/rebase workflows are perhaps one of its most distinguishing features, but they are too complicated for this short guide.

Command Reference

- git init [−−bare] - Initialize a new git repo in a directory
- git status - Print the current status of the local repo
- git config - Get/set global and local configuration
- git add - Add files to the staging area
- git commit [−a] −m <message> - Commit staged changes
- git log [−−pretty=<fmt>] - Print commit history
- git diff [commit] - Create a diff between two points
- git checkout <commit> - Switch to a specific commit
- git remote add <name> <url> - Add a remote to the local config
- git push [−−set−upstream ...] - Push to upstream (setting it if needed)

- git fetch [remote] - Fetch status of a remote
- git pull - Pull from upstream
- git reset - Remove files from staging area
- git stash [list |show|pop] - Manipulate stashes

Configuration Management with Ansible

Overview

Step 1: Setting up and exploring your git/Ansible environment

Your git workflow

We are providing an existing Ansible git repository that will be the base of your configuration management for you cluster. Our first step is to access the "CM" (configuration management server).

The CM lives at 10.0.52.146. You may recognize this as also being our repo server for our yum repos.

You should be able to login with:

```
$ ssh <username>@10.0.52.146
Password: ChangeMe123
[lowell@te-cm ~]$
```

If you take a look at your groups:

```
[lowell@te-cm ~]$ groups
lowell cscnsi group1
```

We will use the cscnsi group to access files that you need. Your group[0−9] is the identifier for your particular group. *You will be working on configuration management as a group.* Note that you do not have sudo access on this system.

You also have access to a group-shared folder:

```
[lowell@te-cm ~]$ ls -l /home/share
total 0
drwxrws--- 2 root group1 6 Jun 12 11:50 group1
drwxrws--- 2 root group2 6 Jun 12 11:50 group2
drwxrws--- 2 root group3 6 Jun 12 11:50 group3
drwxrws--- 2 root group4 6 Jun 12 11:50 group4
```

You will use these folders to host your *bare* ansible repos that you will collaborate with.

The master repo that you will base your cluster on is located at /data/ansible. We will *bare* clone the repo into our group folders:

```
[lowell@te-cm ~]$ cd /home/share/group1
[lowell@te-cm group1]$ git clone --bare /data/ansible
Cloning into bare repository 'ansible.git'...
done.
[lowell@te-cm group1]$ ls -l
total 0
drwxrwsr-x 7 lowell group1 138 Jun 12 12:31 ansible.git
```

Now, as our individual users, we should make our own clone of this repo in our own folder.

```
[lowell@te-cm group1]$ cd
[lowell@te-cm ~]$ git clone /home/share/group1/ansible.git ansible
Cloning into 'ansible'...
done.
[lowell@te-cm ~]$ ls -l
total 0
drwxrwxr-x  5 lowell lowell 85 Jun 12 12:33 ansible
```

Now would be a good time to set your git global config:

```
[lowell@te-cm ansible]$ git config --global user.email lowell@lanl.gov
[lowell@te-cm ansible]$ git config --global user.name "J. Lowell Wofford"
```

Your workflow will be:

1. Pull down the latest version of your own git repo with git pull
2. Make changes you need
3. Push your changes (to /home/share/group*/ansible)
4. (maybe) run your ansible changes

Each group will now be working with their own shared ansible repository.

Navigating your ansible repo

Let's look inside our local clone of the ansible repo.

```
[lowell@te-cm ansible]$ ls -l
total 4
drwxrwxr-x  4 lowell lowell  54 Jun 12 12:33 inventories
drwxrwxr-x 19 lowell lowell 284 Jun 12 12:33 roles
-rw-rw-r--  1 lowell lowell 540 Jun 12 12:33 site.yaml
```

These files have the following purpose:

- inventories defines what hosts we know about and any host/group variables in our configuration.

- roles is where all of our ansible roles live.
- site.yaml is our playbook.

Let's look at each of these.

inventories Under inventories we see:

```
[lowell@te-cm ansible]$ cd inventories/
[lowell@te-cm inventories]$ ls -l
total 4
drwxrwxr-x 2 lowell lowell 62 Jun 12 12:33 group_vars
drwxrwxr-x 2 lowell lowell 23 Jun 12 12:33 host_vars
-rw-rw-r-- 1 lowell lowell 63 Jun 12 12:33 hosts
```

- host_vars is where host specific variables live. This is where you will do most of your work.
- group_vars provide handy variables that are common across all of our clusters. Values set here can be overriden by host_vars.
- hosts is the listing of hosts in our CM. It also binds hosts to groups, which then map to files under group_vars.

Let's take a look at hosts:

```
[lowell@te-cm inventories]$ cat hosts
[masters]
ex-master

[warewulf]
ex-master

[cscnsi]
ex-master
```

Taking a look at group_vars:

```
[lowell@te-cm inventories]$ ls -l group_vars/
total 16
-rw-rw-r-- 1 lowell lowell  132 Jun 12 12:33 cms
-rw-rw-r-- 1 lowell lowell 1643 Jun 12 12:33 cscnsi
-rw-rw-r-- 1 lowell lowell  104 Jun 12 12:33 masters
-rw-rw-r-- 1 lowell lowell  200 Jun 12 12:33 warewulf
```

We can see that the names of these files map to groups in hosts. Variables set in here will be defined for any host that is a member of that group. Let's take a look at one file. Take a look at the cscnsi file under group_vars. Many of these settings should be recognizable.

Under host_vars:

```
[lowell@te-cm inventories]$ ls -l host_vars/
total 4
-rw-rw-r-- 1 lowell lowell 1830 Jun 12 12:33 ex-master
```

We see there is a file that maps in name to values in hosts. The provided ex−master is intended to function as an example for your own configuration.

roles Under roles we find all of the roles we have defined:

```
[lowell@te-cm inventories]$ cd ../roles
[lowell@te-cm roles]$ ls
cluster_lan  common  hosts     mellanox  mount-hyperv-data
 ↪  ohpc_dev_components  powerman  serve-repos  warewulf
cm_pkgs      conman  iptables  modules   munge             opensm
 ↪      repos     slurm
```

We can see there are quite a few of these. A production system might have quite a few more still. The roles tend to be named in a useful way that says what they do. Having a common role is a common convention where tasks that all systems are expected to run go.

We will explore roles in a later exercise where we will write our own.

sites.yml If we look inside sites .yml we see:

```
- hosts: masters
  roles:
    - { role: common, tags: [ 'common', 'base' ] }
    - { role: cluster_lan, tags: [ 'cluster_lan', 'base' ] }
    - { role: mellanox, tags: [ 'mellanox' ] }
    - { role: powerman, tags: [ 'powerman' ] }
    - { role: conman, tags: [ 'conman' ] }
    - { role: slurm, tags: [ 'slurm' ] }
    - { role: warewulf, tags: [ 'warewulf' ] }
    - { role: ohpc_dev_components, tags: [ 'ohpc_dev_components', 'pe' ] }
    - { role: modules, tags: [ 'modules', 'pe' ] }
    - { role: iptables, tags: [ 'iptables' ] }
```

This is a playbook. It links hosts in the group masters to this sequence of roles. We have also asigned tags to each role. This will later give as a convenient way to run ansible with only specific roles.

Step 2: Setting your specific host vars

The first thing you need to do is let me know what your hostname and IP are. I will need to add entries to /etc/hosts on the CM that will map your master hostname to the correct IP.

Once your entry is entered, try pinging your master by hostname:

```
[lowell@te-cm ~]$ ping te-master
PING te-master (172.16.1.254) 56(84) bytes of data.
64 bytes from te-master (172.16.1.254): icmp_seq=1 ttl=64 time=0.957 ms
64 bytes from te-master (172.16.1.254): icmp_seq=2 ttl=64 time=0.510 ms
```

This is important because Ansible will need to be able to access your system by hostname.

Setting your host vars

This repository has everything you need for a base operating system, but you have to tell it about your system specifics.

The following should only be done by one user; follow along/work together in tmux

Go to $HOME/ansible/inventories/host_vars. We need to make a copy of ex−master to your master's hostname. In the examples I will be using the hostname te−master.

```
[lowell@te-cm ~]$ cd $HOME/ansible/inventories/host_vars
[lowell@te-cm host_vars]$ ls
ex-master
[lowell@te-cm host_vars]$ git -v ex-master te-master
'ex-master' -> 'te-master'
```

Now edit this file to have the specific settings for your cluster.

You will need to set the following values at least:

- cluster_prefix is the prefix you want to give your nodes. Up until now we have called them n<number>. In this case the prefix is 'n'. It can be whatever you like.
- cluster_sms_hostname is the hostname of your master. This should be the same as the name of the file your editing.
- cluster_compute_nodes is a "dictionary" of information about yoru compute nodes. You will need to set both mac: and bmc_mac: with your em1 MAC addresses and BMC MAC addresses respectively.
- For cluster_compute_nodes make sure the node names (e.g. ex01) match your cluster_prefix.

You will want to SSH to your currently installed master to get your MAC addresses.

On your master, to get the em MAC addresses:

```
[root@te-master ~]# wwsh node list
NAME                GROUPS              IPADDR              HWADDR
============================================================================
 ↪ =====
n01                 compute             172.16.0.1
 ↪ 18:66:da:ea:34:7c
```

```
...
```

To get just the MAC addresses:

```
[root@te-master ~]# wwsh node list | grep compute | awk '{print $NF}'
18:66:da:ea:34:7c
...
```

To get the BMC MAC addresses:

```
[root@te-master ~]# grep host /etc/warewulf/dhcpd-template.conf
host n01-bmc { hardware ethernet 18:66:da:68:3e:40; fixed-address
...
```

Make sure these are sequential. If they are you can get just a list of the BMC MAC addresses:

```
[root@te-master ~]# grep host /etc/warewulf/dhcpd-template.conf | awk
 ↪  '{print $6}' | sed -e 's/;//'
18:66:da:68:3e:40
```

Save the file. Now commit it to git.

```
[lowell@te-cm ansible]$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#   inventories/host_vars/te-master
nothing added to commit but untracked files present (use "git add" to track)
```

```
[lowell@te-cm ansible]$ git commit -m "added te-master host variables"
[master 1a11af9] added te-master host variables
 1 file changed, 86 insertions(+)
 create mode 100644 inventories/host_vars/te-master
```

We should see that we're one commit ahead of our origin:

```
[lowell@te-cm ansible]$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)
#
nothing to commit, working directory clean
```

Let's not push just yet.

We also need to add our hostname to the inventories/hosts file. We can remove ex−master while we're at it:

```
[masters]
te-master
```

```
[warewulf]
te-master

[cscnsi]
te-master
```

Now let's get that committed to git:

```
[lowell@te-cm ansible]$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 2 commits.
#   (use "git push" to publish your local commits)
#
nothing to commit, working directory clean
```

```
[lowell@te-cm ansible]$ git push
...
Counting objects: 13, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.23 KiB | 0 bytes/s, done.
Total 9 (delta 3), reused 0 (delta 0)
To /home/share/group1/ansible.git
   403ff07..718eb3b  master -> master
```

```
[lowell@te-cm ansible]$ git status
# On branch master
nothing to commit, working directory clean
```

Our origin should now be up-to-date.

One of the other users should `git pull` on their clone and make sure they see the changes.

Step 3: Re-install the master

We now need to go re-install our master to have a clean slate.

IMPORTANT: Make sure you have all of your MAC addresses before we do this!

Step 4: Deploy with Ansible

Setup SSH keys

Every user on `te-cm` should generate SSH keys. Run:

```
[lowell@te-cm ~]$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/lowell/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/lowell/.ssh/id_rsa.
Your public key has been saved in /home/lowell/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:3tVMkpmfyxky7P5BDxCvXqzgjCs3Y0hPBo+qtD7srNw lowell@te-cm
The key's randomart image is:
+---[RSA 2048]----+
|            .    |
|             *   |
|            * o  |
|      .    . X . |
|       +S . * @  |
|      o.+= = B * |
| ..  o =o + + = .|
|.o+.. o *. .   . |
|.=*E   +.o  ...  |
+----[SHA256]-----+
```

Now, grab the contents of $HOME/.ssh/id_rsa.pub for each user:

```
[lowell@te-cm ~]$ cat $HOME/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqmS0Aoxn⌟
 ↪  mynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FVE2ffKg⌟
 ↪  hpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+gnpUiQR⌟
 ↪  a5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3CiRsBKS0⌟
 ↪  4Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNTw+pc1X⌟
 ↪  fzQPcfit8+k1QJf5NOHL5
 ↪  lowell@te-cm
```

SSH to the master using your admin account. Become root and add these keys to /root/.ssh/authorized_keys. Make sure the permissions on /root/.ssh/authorized_keys are set to 0600.

```
$ ssh admin@master
$ sudo -i
# ssh-keygen
...
# vi $HOME/.ssh/authorized_keys
(add your keys, line-by-line)
# chmod 0600 $HOME/.ssh/authorized_keys
```

Your users should now be able to ssh from the CM to your master using ssh root@<hostname> without a password. We will use this for running ansible.

Running ansible

We will use the ansible−playbook command to run ansible. We will do this *in* your $HOME-/ansible directory as yourself. You should be careful not to have more than one person running ansible at once.

The options we will use for ansible−playbook are:

```
ansible-playbook -u root -i inventories/hosts -l <hostname> -t <tags>
 ↪  site.yml
```

The −u root option tells ansible to ssh to the master as root.

If we don't specify −l ansible will run on all hosts.

If we don't specify −t ansible will run on all defined roles in the site .yml.

It's a good idea to run just the "common" tag until everything is working.

Try running on the common tag:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t common site.yaml

PLAY [masters] ************************************************************↲
 ↪  ******************************

TASK [Gathering Facts] ****************************************************↲
 ↪  **************************************

...(lots of output)

PLAY RECAP ****************************************************************↲
 ↪  **************************
te-master                 : ok=15   changed=7    unreachable=0    failed=0
```

If all went well, the summary should say failed =0.

We should now be able to fully deploy our master:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master site.yaml | tee $HOME/ansible.log
...(huge amount of input; much time)
```

Note:

- We didn't specify −t, so this will run the whole recipe.
- We tee'ed output to $HOME/ansible.log so we can review the output later.

In the ideal world, you would have a perfectly deployed system at this point. Unfortunately, there is a known bug in the warewulf role where we probably need to run it twice. Let's run just that role again:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
↪  te-master -t warewulf site.yaml | tee $HOME/ansible-warewulf.log
...(huge amount of input; much time)
```

Our system should now be deployed with only a couple of missing pieces. We will fix those mising pieces (passwd files, ntp, etc) next time by writing our own extension roles.

Check our master

Let's check our master to see what's working.

Check:

- Are the right services running? (dhcpd, xinetd, httpd, powerman, conman, slurmctld)
- What does wwsh node list say?
- What does pm −q say?
- Can you conman to a node?
- Can you boot a node?
- Do the interfaces look ok on the node? eth0? ib0?
- Can you run IB fabric tests on the node? Does everything look OK?
- Does Slurm look ok? What does sinfo say? Do you need to resume node(s)?
- Can you do an salloc to get to a node?
- Can you see lmod packages on a node with module avail?

If all of this checks out, you're good to go! Go ahead and boot all of your nodes.

Next time we'll learn more about how to write our own ansible roles to configure more.

Challenge: Install fortune−mod on your nodes using ansible.

Configuration Management with Ansible: Using Ansible

Overview

In the previous part, we learned to use Ansible to deploy a cluster. In this part, we will add some functionality to our cluster with new Ansible roles.

Step 1: Writing a role to add our users, first try

Designing the "users" role

You probably already noticed that one of the things that our ansible does not do is set up our individual user accounts. Let's get started by making a role to do this. There are several ways to go about this, but let's start with the simplest to implement first.

One we can create the user accounts is to have ansible provide the whole of the {passwd,group,shadow} files directly to the master. There are some obvious pitfalls to this, but it should get us started. Note for instance that storing our shadow file in our config repo with password hashes is probably not extremely secure. Let's do it this way to start out anyway.

As we saw in the last part, we can find roles for our repo in $HOME/ansible/roles. We'll be creating a new one of these. It's a good idea to give it a name that is simple yet descriptive. We'll call our role users.

We can start by making the directory for the role as well as it's tasks folder. The tasks folder is where our plays live for our role. By default ansible will load the main.yml file under tasks, but the main.yml file could, for instance, load other YAML files of tasks.

```
[lowell@te-cm roles]$ mkdir -p users/tasks
[lowell@te-cm roles]$ cd users/tasks/
```

The "users" tasks

We'll need to create our tasks file. It will have to have three tasks in it, one for each of the files we intend to copy to the master.

Ansible tasks are performed by ansible modules. Our first job is figuring out which module to use. There are a lot of them available. Take a look at The Ansible Module Index. This is your source for what modules available, along with documentation on how to use them.

We're trying to copy files to the master and put them in place (with appropriate ownership, permissions, etc). The module that makes sense for this is the "copy" module, see copy module documentation. Using the "copy" module we can write our three simple tasks:

```
---
- name: Setup /etc/passwd
  copy:
    src: passwd
    dest: /etc/passwd
    owner: root
    group: root
    mode: '0644'

- name: Setup /etc/group
  copy:
    src: group
    dest: /etc/group
    owner: root
    group: root
    mode: '0644'

- name: Setup /etc/shadow
  copy:
    src: shadow
    dest: /etc/shadow
    owner: root
    group: root
    mode: '0000'
```

Note that we have assigned specific ownership and permissions to each file. This prescriptive definition is at the heart of why we use configuration management. Ansible will try to ensure that all of these aspects are correct every time it runs.

Also, notice that we gave each task a descriptive name. There are a lot of naming conventions people use for Ansible, but it is universally agreed that any best practice gives a short, descriptive name for what the play does.

Now that we have the main.yml file setup, we need to actually provide the files to copy in. Just like tasks is a special folder for plays in the role, the files directory can contain files to be provided through modules like the "copy" module. We just need a passwd, group and shadow file to put here.

First, make the files directory, then we can scp the necessary files from our master:

```
[lowell@te-cm tasks]$ cd $HOME/ansible/roles/users/
[lowell@te-cm users]$ mkdir files
[lowell@te-cm users]$ cd files
[lowell@te-cm files]$ scp root@te-master:/etc/{passwd,group,shadow} .
passwd
 ↪            100% 1500   564.5KB/s   00:00
group
 ↪            100%  644   259.8KB/s   00:00
shadow
 ↪            100%  932   159.1KB/s   00:00
[lowell@te-cm files]$ ls
group  passwd  shadow
```

Our role is now complete. If we update these files, those updates will propagate to our master.

Adding the role to the playbook & running

We have a role now, but we need it to actually be added to our playbook for it to work. We do this by editing site.yml. We can add a line to site.yml like the one below:

```
...
    - { role: common, tags: [ 'common', 'base' ] }
    - { role: users, tags: [ 'users', 'base' ] }
    - { role: cluster_lan, tags: [ 'cluster_lan', 'base' ] }
...
```

That’s all there is to it. We can now try out our play by running:

NOTE: This is a bit dangerous. Be very careful not to accidentally mangle these files, or you may have to work hard to get back into your master.

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t users site.yaml

PLAY [masters] ************************************************************⌟
 ↪  *******************

TASK [Gathering Facts]
 ↪  **********************************************************************
ok: [te-master]

TASK [users : Setup /etc/passwd]
 ↪  ************************************************************
ok: [te-master]


...
PLAY RECAP ****************************************************************⌟
 ↪  *******************
te-master                  : ok=4    changed=0    unreachable=0    failed=0
```

It seems to have run fine, but nothing changed (because the files should be the same). Let’s add a group, “testers”, to roles/users/files/group to try it out:

```
...
cgred:x:994:
tss:x:59:
testers:x:65533:
```

Now run it again:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t users site.yaml

PLAY [masters] ************************************************************⌟
 ↪  *******************
...

TASK [users : Setup /etc/group]
 ↪  *************************************************************
changed: [te-master]
```

```
...
PLAY RECAP ****************************************************************
 ↪  ******************
te-master                  : ok=4    changed=1    unreachable=0    failed=0
```

Ok, it reported that we got a change. Let's see if it took. We can ssh to the master and check:

```
[lowell@te-cm ansible]$ ssh root@te-master tail -n3 /etc/group
cgred:x:994:
tss:x:59:
testers:x:65533:
```

That worked!

But, this is clearly less than ideal. For one, we have to specify the *whole* {passwd,group,shadow} files. Also, we have to store secrets in our repository, which doesn't seem like a good idea.

Step 2: Writing an NTPD role

Designing the "ntp" role

You may have noticed that our cluster does not have the NTPD service defined. This is an important service, and we'll want it for a fully configured system. We can write a play that will create the service.

First, we need to create the role and tasks folder:

```
[lowell@te-cm roles]$ mkdir -p ntp/tasks
[lowell@te-cm roles]$ cd ntp/tasks
```

Let's make a list of the things we'll need to do to get NTP running:

1. NTP needs a correct timezone; we should probably set that first
2. We need to make sure the package is installed
3. We need to write its config: /etc/ntp.conf
4. We need to enable the ntpd and ntpdate services

Setting the timezone

Let's start with the timezone task. Looking through the module list we see that there's a timezone module. That will make the first item easy. We can start our main.yml with:

```
---
- name: set timezone
  timezone:
    name: "America/Denver"
```

That's easy enough, but it would be nice if we didn't have to hard-code the timezone. Fortunately, we don't. We can make the timezone a variable. All we have to do is change this to:

```
---
- name: set timezone
  timezone:
    name: "{{ ntp_timezone }}"
```

Now ansible will set the timezone to the value of the variable ntp_timezone. That variable can be set in a lot of places, like the host_vars or the group_vars.

Installing the ntp package

Moving on to the second item, we can use the package module. The package module provides a way to install packages that isn't package-manager specific. There are also ways to specifically use, say, yum, but we don't need any special features, so this module will work. Our play for this is relatively simple:

```
- name: install ntpd
  package:
    name: ntp
    state: present
```

The format of this is pretty straight forward. We give it the name of the package, ntp, and say that we want it to be present. That's it.

Creating `ntp.conf`

Next we need to configure NTP. A very simple NTP config might look like this:

```
driftfile /var/lib/ntp/drift
restrict default nomodify notrap nopeer noquery
restrict 127.0.0.1
restrict ::1
restrict 172.16.0.0 mask 255.255.255.0 nomodify notrap
server 10.0.52.146 prefer
server 127.127.1.0
fudge  127.127.1.0 stratum 10
includefile /etc/ntp/crypto/pw
keys /etc/ntp/keys
disable monitor
```

We could use the "copy" module and copy this file in as we did with the files in Step 1. But, it would be nice to be more flexible. It would be nice to not fully specify the internal network (172.16.0.0/24), or the server (10.0.52.146) and instead provide those as variables.

To achieve this, we can use the template module. The template module works a bit like copy, but it will fille Jinja2 templates out instead of just copying simple files. The Jinja2 template can use variables much like the variables used in ansible plays themselves. We can use:

```
driftfile /var/lib/ntp/drift
restrict default nomodify notrap nopeer noquery
restrict 127.0.0.1
restrict ::1
restrict {{ ntp_allow_net }} mask {{ ntp_allow_netmask }} nomodify notrap
server {{ ntp_server }} prefer
server 127.127.1.0
fudge  127.127.1.0 stratum 10
includefile /etc/ntp/crypto/pw
keys /etc/ntp/keys
disable monitor
```

Note the specific values are now variable values, like {{ ntp_allow_net }}. We can specify these in the host_vars, group_vars, and other places.

The play for this would be:

```
- name: update ntpd.conf
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    owner: root
    group: root
    mode: 0444
```

As we can see, this looks a lot like copy.

The template itself lives in the ntp/templates folder. Make this directory and create the file ntp.conf.j2 with the template contents above.

Enabling/starting the ntp services

The last of our requires is enabling the ntpd and ntpdate services. We can use the systemd module. This module lets us manage the various states of systemd services.

```
- name: enable ntp service
  systemd:
    name: ntpdate
    enabled: yes
    state: started

- name: enable ntpdate service
  systemd:
    name: ntpdate
    enabled: yes
```

We simply tell systemd to make sure those two services are enabled, and that they've been started. Note that we only want to tell systemd to enable ntpdate, but not start it. ntpdate is intended to be run on startup only.

Setting up defaults

If we tried to run our plays without defining the variables we use, they would fail. It would be nice to make sure the variables are at least always defined. We can do this in the ntp/defaults folder. Create that folder, and add a file called main.yml with these contents:

```
---
ntp_timezone: "America/Denver"
ntp_server: "10.0.52.146"
ntp_allow_net: "172.16.0.0"
ntp_allow_netmask: "255.255.255.0"
```

This not only makes sure that some sensible defaults are assigned and the play won't fail but is a place we can look to see what variables the role uses at a glance.

Creating an ntpd handler

There's one last step to making this role complete. We would really like to make sure that any time the ntp.conf gets updated ntpd gets restarted. This can be achieved with what ansible calls handlers. Handlers are just plays, but they don't run by default. If a play in the tasks folder gets marked as a change, and it has a specification notify : <handler_name>, then the handler will get run at the very end of the ansible run. The handler will only get run once no matter how many times it is called.

Handlers live in the ntp/handlers folder. Let's create a file main.yml in that folder and add the following:

```
---
- name: restart ntpd
  systemd:
    name: ntpd
    state: restarted
```

As you can see, we've used the systemd module. This module lets us manage the various states of systemd services.

Now, let's modify our ntp.conf play from above and add a notify ::

```
- name: update ntpd.conf
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
    owner: root
    group: root
    mode: 0444
  notify: restart ntpd
```

Notice that we need restart ntpd to be exactly the value of notify: as well as the name: of the handler.

Putting it all together

Our role should now be complete. The filesystem layout should look like this:

```
[lowell@te-cm roles]$ tree ntp
ntp
|-- defaults
|   `-- main.yml
|-- handlers
|   `-- main.yml
|-- tasks
|   `-- main.yml
`-- templates
    `-- ntp.conf.j2

4 directories, 4 files
```

We can enable the role in the playbook. Edit the site.yml and add it there:

```
...
    - { role: cluster_lan, tags: [ 'cluster_lan', 'base' ] }
    - { role: ntp, tags: [ 'ntp', 'base' ] }
    - { role: mellanox, tags: [ 'mellanox' ] }
...
```

Now, let's run it:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t ntp site.yaml

PLAY [masters] ************************************************************⌋
 ↪  ******************

TASK [Gathering Facts]
 ↪  ***********************************************************************
ok: [te-master]
```

```
TASK [ntp : set timezone]
 ↪  ******************************************************************
ok: [te-master]
...
```

Let's login to the master and make sure it's actually working:

```
[lowell@te-cm ansible]$ ssh root@te-master
Last login: Thu Jun 13 00:40:29 2019 from 172.16.1.252
tp[root@te-master ~]# ntpq
ntpq> lpeer
     remote           refid      st t when poll reach   delay   offset
 ↪  jitter
===========================================================================⌋
 ↪  ===
 te-hyperv       .INIT.          16 u   41   64    0    0.000    0.000
 ↪  0.000
*LOCAL(0)        .LOCL.          10 l   40   64    1    0.000    0.000
 ↪  0.000
ntpq>
```

Looks like it's working.

Step 3: Writing a role to add our users, take two

(Re)designing the users role

In Step 1 we wrote a simple "users" module, but it's not very flexible, secure or maintainable. Let's try a better way now that we've learned a couple more tools. We can look at the module list, and notice there's a module called "users" that can do a lot of what we need. Moreover, it might be nice to auto-setup public key authentication rather than pass around password hashes. There's an "authorized_keys" module for that.

In general, it's a good idea to use modules that are built for our task rather than run commands (see command module) or directly manipulate/copy files. This will ultimately be more portable and future-proof since it leverages the work of developers that are actively maintaining the module.

To get a clean slate, let's move the old role out of the way and create a new one:

```
[lowell@te-cm roles]$ mv users users-simple
[lowell@te-cm roles]$ mkdir -p users/tasks
```

The “users” and “authorized_keys” modules

In our tasks/main.yml file, we can add an arbitrary user named “joe” and add an authorized key with a play like this (feel free to substitute real information for a user):

```
---
- name: Add the user 'joe', make admin
  user:
    name:   joe
    shell:  /bin/bash
    groups: wheel
    append: yes

- name: Set authorized key for joe
  authorized_key:
    user: joe
    state: present
    key: |
      ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqm ⌋
 ↪  SOAoxnmynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FV ⌋
 ↪  E2ffKghpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+g ⌋
 ↪  npUiQRa5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3Ci ⌋
 ↪  RsBKS04Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNT ⌋
 ↪  w+pc1XfzQPcfit8+k1QJf5NOHL5
 ↪  lowell@te-cm
```

We already have our role enabled. Let’s give it a try:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t users site.yaml
...

TASK [users : Add the user 'joe', make admin]
 ↪  *************************************************
changed: [te-master]

TASK [users : Set authorized key for joe]
 ↪  *****************************************************
changed: [te-master]


...
te-master                  : ok=3    changed=2    unreachable=0    failed=0
```

Now, let’s see if it worked.

```
[lowell@te-cm ansible]$ ssh root@te-master
Last login: Thu Jun 13 05:27:49 2019 from 172.16.1.252
```

```
[root@te-master ~]# id joe
uid=1001(joe) gid=1001(joe) groups=1001(joe),10(wheel)
[root@te-master ~]# cat /home/joe/.ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqmSOAoxn
↪  mynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FVE2ffKg
↪  hpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+gnpUiQR
↪  a5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3CiRsBKS0
↪  4Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNTw+pc1X
↪  fzQPcfit8+k1QJf5NOHL5
↪  lowell@te-cm
```

Looks like it did.

Using dictionaries & loops

Of course, this is a very static play. It would be much nicer if we could configure our users and keys like we configure everything else.

We can use “dictionaries” and “loops” in ansible to do this. We’ve already seen dictionaries. Our cluster node definitions are dictionaries.

It often helps to design an example dictionary entry first, then develops the play from that. We need to be able to specify in each entry:

- the username
- any add-on groups (like wheel)
- the ssh-key

In a more complete role we might also allow for specifying things like the shell too, but let’s keep this simple.

Any example dictionary might look like:

```
cluster_users:
  joe:
    groups: wheel
    sshkey: |
      ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqm
↪  SOAoxnmynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FV
↪  E2ffKghpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+g
↪  npUiQRa5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3Ci
↪  RsBKS04Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNT
↪  w+pc1XfzQPcfit8+k1QJf5NOHL5
↪  lowell@te-cm
```

Go ahead and add a dictionary like this to your inventories/host_vars/<hostname> host vars file. You should use real information for your users.

We could then define all of the users we want in this dictionary. How do we design the tasks.yml? We'll start with the answer, then explain it:

```
---
- name: Add the users
  user:
    name:   "{{ item.key }}"
    shell:  /bin/bash
    groups: "{{ item.value.groups }}"
    append: yes
  loop: "{{ query('dict', cluster_users|default({})) }}"

- name: Setup authorized keys
  authorized_key:
    user: "{{ item.key }}"
    state: present
    key: "{{ item.value.sshkey }}"
  loop: "{{ query('dict', cluster_users|default({})) }}"
```

The loop: parameter tells ansible that the play should be run once for each result of the expression it contains. The value of each iteration will be set in item (note: you *can* rename this, but it's often unnecessary). To get the key of the dictionary (i.e. the username), we use "{{ item.key }}". To get a particular value, we use "{{ item.value.<name> }}".

Let's run our role:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts -l
 ↪  te-master -t users site.yaml
...

TASK [users : Add the users]
 ↪  *****************************************************************
changed: [te-master] => (item={'key': u'cluening', 'value': {u'sshkey':
 ↪  u'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqm
 ↪  SOAoxnmynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FV
 ↪  E2ffKghpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+g
 ↪  npUiQRa5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3Ci
 ↪  RsBKS04Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNT
 ↪  w+pc1XfzQPcfit8+k1QJf5NOHL5 lowell@te-cm\n', u'groups':
 ↪  u'wheel'}})

TASK [users : Setup authorized keys]
 ↪  *********************************************************
```

```
changed: [te-master] => (item={'key': u'cluening', 'value': {u'sshkey':
 ↪  u'ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqm⌋
 ↪  SOAoxnmynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FV⌋
 ↪  E2ffKghpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+g⌋
 ↪  npUiQRa5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3Ci⌋
 ↪  RsBKS04Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNT⌋
 ↪  w+pc1XfzQPcfit8+k1QJf5NOHL5 lowell@te-cm\n', u'groups':
 ↪  u'wheel'}})

PLAY RECAP ****************************************************************⌋
 ↪  *******************
te-master                  : ok=3    changed=2    unreachable=0    failed=0
```

In my case, I added the "cluening" user. I gave it the ssh key owned by the current user, so I should be able to ssh to that user on the master. Let's try it out:

```
[lowell@te-cm ansible]$ ssh cluening@te-master
Configuring SSH for cluster access
[cluening@te-master ~]$ cat .ssh/authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD5jy8UVjx2etSeDMnpC91t30R0Z8dqmSOAoxn⌋
 ↪  mynNpjbRiMrZHYe1jvBS+2ERD0D9aZu2iOfHsc9cq4LzdyKMrYIhvdVfqMTQI68FVE2ffKg⌋
 ↪  hpwT0IRfYn+3sjLc/NxH7Pnra+IzXk81BntISkjmqp7wavDMH6k3Dw8kTYTaedD+gnpUiQR⌋
 ↪  a5EunQaO2xrPxEr2XFv+KmY+qBiagIDg1W/rQbDxDxpE5QmBOM3EFM7zvv+DUp3CiRsBKS0⌋
 ↪  4Q9edCxp/ceSK0mDXbVZtDW9Op2b58IcKOOupQXXt6ax0nzn7CksglvaTkzIxZMNTw+pc1X⌋
 ↪  fzQPcfit8+k1QJf5NOHL5
 ↪  lowell@te-cm
...
```

We now have a much safer, more secure, and more sustainable way to add our users and give them access. This is hardly complete but will work for our purposes. For instance, it doesn't handle *deleting* unspecified users.

Make sure you commit all of these changes to your git repo.

Monitoring with `rsyslog` and Splunk

Overview

Monitoring for HPC is a big topic. Many larger HPC groups have whole teams dedicated to monitoring. There are many types of monitoring, many scalability issues and many different kinds of things you may want to monitor for. There are also many different tools available.

We are going to focus on monitoring through system logs. We will setup an ansible play that will aggregate system logs to the master from all of the nodes. We will then setup the (commercial) tool, Splunk, to provide us with analytics based on these system logs.

Step 1: Setting up `rsyslog`

rsyslog is a service that takes system log messages and writes them to the log files that we are accustomed to seeing in /var/log. rsyslog also supports re-transmitting log messages to another host.

In our clusters, we are going to tell rsyslog on the nodes to aggregate all of its log messages on the master. This will give us a convenient central location to diagnose and monitor all of our system logs.

Creating the rsyslog role (master)

Let's create our role structure:

```
[lowell@te-cm tasks]$ cd $HOME/ansible/roles
[lowell@te-cm roles]$ mkdir -p rsyslog/tasks
[lowell@te-cm roles]$ cd rsyslog/tasks
```

We will be following a design pattern similar to what we have used in the "Writing Ansible" guide when we created the "ntp" role. Our steps will be:

1. make sure rsyslog is installed
2. generate our rsyslog.conf file
3. make sure rsyslog is enabled and running

Make tasks/main.yml with:

```
---
- name: ensure syslog is installed
  package:
    name:
      - rsyslog
    state: present

- name: master rsyslog.conf file
  template:
    src: master-rsyslog.conf.j2
    dest: /etc/rsyslog.conf
    owner: root
    group: root
    mode: 0444
    backup: yes
  notify: restart rsyslog

- name: ensure syslog is running
  systemd:
    name: rsyslog
    state: started
    enabled: yes
```

Notice that we created a restart rsyslog handler. Make the file handlers/main.yml with:

```
---
- name: restart rsyslog
  systemd:
    name: rsyslog
    state: restarted
```

So far, this should all look very similar to the ntp role.

Now we need our template. Make the file templates/master−rsyslog.conf.j2 with:

```
$ModLoad imuxsock # provides support for local system logging (e.g. via
 ↪  logger command)
$ModLoad imjournal # provides access to the systemd journal
$ModLoad imudp
$UDPServerRun 514
$WorkDirectory /var/lib/rsyslog
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
$IncludeConfig /etc/rsyslog.d/*.conf
$OmitLocalLogging on
$IMJournalStateFile imjournal.state
*.info;mail.none;authpriv.none;cron.none                /var/log/messages
authpriv.*                                              /var/log/secure
mail.*                                                  -/var/log/maillog
cron.*                                                  /var/log/cron
*.emerg                                                 :omusrmsg:*
uucp,news.crit                                          /var/log/spooler
local7.*                                                /var/log/boot.log
```

Notice that this is setup as a template, but it actually doesn't have any variables in it. We could have done this with the copy module. However, if we want to add variables to this file down the road, it will prove convient to already have it as a template. This is a common practice with files that we think may need template variables some day.

Creating the rsyslog role (node)

This covers the configuration for the master, but now we need the configuration for the node. There are several ways we could handle this, but we will handle it be directly manipulating the BOS for warewulf.

Note: this means that rsyslog must be after warewulf in the site .yml file.

Let's add the following tasks to the end of the tasks/main.yml file:

```
- name: ensure that syslog is installed in BOS
  yum:
    name: rsyslog
    state: present
    installroot: "{{ item.path }}"
  loop: "{{ cluster_bos_images }}"
  notify:
    - rebuild vnfs
    - rebuild image

- name: syslog, rsyslog.conf file
  template:
    src: compute-rsyslog.conf.j2
    dest: '{{ item.path }}/etc/rsyslog.conf'
    owner: root
    group: root
    mode: 0444
    backup: yes
  loop: '{{ cluster_bos_images }}'
  notify:
    - rebuild vnfs
```

These two require a little explanation. First of all, cluster_bos_images can be found in your inventory/group_vars. It defines information about each image you build for nodes. We actually have support for multiples, but we're only using one.

You'll notice that we used the yum module, and not the package module. The yum module allows us to use installroot :, which will work with packages in our BOS.

Finally, notice that this template gets installed inside our BOS. We don't want to add this to the warewulf synced files, because we expect this file is essentially static once setup. Instead, we "bake it in" to the image, i.e. it is permanently resident in our VNFS.

But, since both of these tasks change the image, we need to make sure the vnfs gets rebuilt. The warewulf role adds the handler, rebuild vnfs, to make sure that happens.

Now we need the templates/compute−rsyslog.conf.j2 template:

```
$ModLoad imuxsock # provides support for local system logging (e.g. via
↪  logger command)
$ModLoad imjournal # provides access to the systemd journal
$WorkDirectory /var/lib/rsyslog
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
$IncludeConfig /etc/rsyslog.d/*.conf
$OmitLocalLogging on
$IMJournalStateFile imjournal.state
*.emerg                                                 :omusrmsg:*
local7.*                                                /var/log/boot.log
```

```
{{ syslog_compute_remote_host }}
```

The format for what goes in the syslog_compute_remote_host line is:

```
*.* @<host_ip>:<host_port>
```

Let's setup an appropriate default in defaults/main.yml:

```
---
syslog_compute_remote_host: "*.* @{{ cluster_sms_ip }}:514"
```

The default port for rsyslog is 514. If you look in inventories/group_vars/cscnsi, you'll see that cluster_sms_ip is already defined.

Setting dependencies and enabling the role

We noticed above that this role assumes that the warewulf role is also run. Ansible gives us a way to declare this dependency. To make the dependency, we need the file meta/main.yml in the role with:

```
---
dependencies:
- role: warewulf
```

This will ensure that anytime we enable rsyslog we also enable warewulf.

Now, let's enable our role in site .yml:

```
...
    - { role: warewulf, tags: [ 'warewulf' ] }
    - { role: rsyslog, tags: [ 'rsyslog' ] }
    - { role: ohpc_dev_components, tags: [ 'ohpc_dev_components', 'pe' ] }
...
```

And run it:

```
[lowell@te-cm ansible]$ ansible-playbook -u root -i inventories/hosts
 ↪  site.yaml -l te-master -t rsyslog
...
PLAY RECAP ****************************************************************
 ↪  *************
te-master                  : ok=65   changed=28   unreachable=0    failed=0
```

Notice that we ran with −t rsyslog, but it first ran the warewulf role as we required by our dependencies.

Let's make sure the right configuration got written into our BOS:

```
[lowell@te-cm ansible]$ ssh root@te-master
Last login: Fri Jun 14 06:33:38 2019 from 172.16.1.252
```

```
[root@te-master ~]# cat
 ↪  /opt/ohpc/admin/images/centos/compute/etc/rsyslog.conf
$ModLoad imuxsock # provides support for local system logging (e.g. via
 ↪  logger command)
$ModLoad imjournal # provides access to the systemd journal
$WorkDirectory /var/lib/rsyslog
$ActionFileDefaultTemplate RSYSLOG_TraditionalFileFormat
$IncludeConfig /etc/rsyslog.d/*.conf
$OmitLocalLogging on
$IMJournalStateFile imjournal.state
*.emerg                                                 :omusrmsg:*
local7.*                                                /var/log/boot.log
*.* @172.16.0.254:514
```

Ok, that looks right. Now we need to reboot our nodes into the new image.

```
[root@te-master ~]# pdsh -w te[01-10] systemctl reboot
te03: Warning: Permanently added 'te03,172.16.0.3' (ECDSA) to the list of
 ↪  known hosts.
te07: Warning: Permanently added 'te07,172.16.0.7' (ECDSA) to the list of
 ↪  known hosts.
te04: Warning: Permanently added 'te04,172.16.0.4' (ECDSA) to the list of
 ↪  known hosts.
te05: Warning: Permanently added 'te05,172.16.0.5' (ECDSA) to the list of
 ↪  known hosts.
...
```

While the nodes are booting, let's tail the /var/log/messages. We should start to see syslog events from our nodes coming through:

```
Jun 14 06:52:39 te09 systemd: Removed slice User Slice of root.
Jun 14 06:52:39 te04 systemd: Removed slice User Slice of root.
Jun 14 06:52:44 te09 ntpd[4741]: 0.0.0.0 c628 08 no_sys_peer
Jun 14 06:52:45 te01 ntpd[3491]: 0.0.0.0 0618 08 no_sys_peer
```

We see the format is:

```
<date> <host> <service>: <message>
```

We can also send a test message from one fo the nodes:

```
[root@te01 ~]# logger "Test Message"
```

Should result in:

```
Jun 12 06:55:18 te01 root: Test Message
```

On the master.

Great! Now we an see all syslog events from the nodes on the master.

Step 2: Splunk

We will now setup Splunk to analyze the data from rsyslog. Much of this step will be interactive exploration, since splunk provides a web GUI.

We are not going to use ansible to set this up. We will install it directly on the master. This is just a trial license of Splunk, and we're just going to install it on the master to play around with it.

First, download the splunk rpm to your master:

```
[root@te-master ~]# curl -O http://10.0.52.146/repo/splunk.rpm
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
 ↪  Current
                                 Dload  Upload   Total   Spent    Left
 ↪  Speed
100  366M  100  366M    0     0   130M      0  0:00:02  0:00:02 --:--:--
 ↪  130M
```

Now install it:

```
[root@te-master ~]# rpm -Uvh splunk.rpm
warning: splunk.rpm: Header V4 RSA/SHA256 Signature, key ID b3cd4420: NOKEY
Preparing...                          #################################
 ↪  [100%]
    package splunk-7.3.0-657388c7a488.x86_64 is already installed
```

This will create /opt/splunk.

```
[root@te-master ~]# ls /opt/splunk
bin            etc      lib                 openssl            share
 ↪                                          var
copyright.txt  include  license-eula.txt  README-splunk.txt
 ↪  splunk-7.3.0-657388c7a488-linux-2.6-x86_64-manifest
```

We have to manually start splunk the first time. It will require us to accept a license agreement, and set a username/password.

```
[root@te-master bin]# cd /opt/splunk/bin
[root@te-master bin]# ./splunk start

...(accept license, etc)

Checking prerequisites...
    Checking http port [8000]: open
    Checking mgmt port [8089]: open
    Checking appserver port [127.0.0.1:8065]: open
    Checking kvstore port [8191]: open
    Checking configuration... Done.
    Checking critical directories...    Done
    Checking indexes...
        Validated: _audit _internal _introspection _telemetry
 ↪  _thefishbucket history main summary
    Done
    Checking filesystem compatibility...  Done
    Checking conf files for problems...
    Done
    Checking default conf files for edits...
    Validating installed files against hashes from
 ↪  '/opt/splunk/splunk-7.3.0-657388c7a488-linux-2.6-x86_64-manifest
    All installed files intact.
    Done
All preliminary checks passed.

Starting splunk server daemon (splunkd)...
Done
                                                           [  OK  ]

Waiting for web server at http://127.0.0.1:8000 to be available... Done


If you get stuck, we're here to help.
Look for answers here: http://docs.splunk.com

The Splunk web interface is at http://te-master:8000
```

Notice that it starts on port 8000 of the master. We don't actually want to start splunk this way though. Let's stop it, and use their command to enable it through systemd:

```
[root@te-master bin]# ./splunk stop
Stopping splunkd...
Shutting down.  Please wait, as this may take a few minutes.
..                                                         [  OK  ]
Stopping splunk helpers...
                                                           [  OK  ]
Done.
[root@te-master bin]# /opt/splunk/bin/splunk enable boot-start
Init script installed at /etc/init.d/splunk.
Init script is configured to run at boot.
[root@te-master bin]# systemctl start splunk
```

Now splunk is managed by systemd, and will start on reboot.

At this point, you should be able to open a browser and go to:

```
http://<your_ip>:8000/
```

Login with your credentials you set for splunk.

From this point we will explore together on the projector

Practical HPC benchmarking

Overview

Benchmarking can be a critical part of system design. Often a system must meet certain benchmarks to meet "acceptance". We will be focused on *synthetic* benchmarks in this guide. A real benchmarking suite should also include some *real* benchmarks, i.e. runs of real programs of interested on well-understood problem sets.

We will go through a handful of common benchmarking techniques through the steps of this guide. We will spend most of the steps working on different micro-benchmarks focused on the performance of sub-components. We will end by running a suite of HPL/Linpack benchmarks that we will analyze.

Step 1: Memory performance benchmarking with Stream

Building and running Stream

Stream is a straight forward and easy to build benchmark. We will use it to get a sense of the memory performance of our nodes.

You can get the Stream benchmark from: https://github.com/jeffhammond/STREAM .

Remember, your nodes probably aren't configured to get directly to the internet, so you should work on your master. It might make sense to set up all of the benchmarking software in your /home/share/group$i directory.

```
[lowell@te-master ~]$ git clone https://github.com/jeffhammond/STREAM
Cloning into 'STREAM'...
remote: Enumerating objects: 54, done.
remote: Total 54 (delta 0), reused 0 (delta 0), pack-reused 54
Unpacking objects: 100% (54/54), done.
[lowell@te-master ~]$ cd STREAM/
[lowell@te-master STREAM]$ ls
HISTORY.txt  LICENSE.txt  Makefile  README  mysecond.c  stream.c  stream.f
```

If you look at the Makefile you'll see it specifically references gcc−4.9 and gfortran−4.9. You can simply change these to gcc and gfortran.

You should be using gcc versio 8.3.0:

```
[lowell@te-master STREAM]$ gcc --version
gcc (GCC) 8.3.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

This is provided by lmod:

```
[lowell@te-master STREAM]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
```

Once the Makefile is modified, you should be able to build without trouble:

```
[lowell@te-master STREAM]$ make
gcc -O2 -fopenmp   -c -o mysecond.o mysecond.c
gcc -O2 -fopenmp -c mysecond.c
gfortran -O2 -fopenmp -c stream.f
gfortran -O2 -fopenmp stream.o mysecond.o -o stream_f.exe
gcc -O2 -fopenmp stream.c -o stream_c.exe
```

This makes two executables: stream_c.exe (C-based) and stream_f.exe (Fortran-based). When I built this, my stream_f was unusable, but stream_c works fine.

Go ahead and run stream_c to test it out:

```
[lowell@te-master STREAM]$ ./stream_c.exe
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
...
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:           16103.3     0.010317     0.009936     0.010613
Scale:          14988.0     0.011003     0.010675     0.011660
Add:            17613.6     0.014080     0.013626     0.014446
Triad:          18186.0     0.013529     0.013197     0.013859
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

You'll notice there are results for four tests:

- “Copy” copies an element of memory: a[i] = b[i]
- “Scale” multiplies an element of memory by a scalar: a[i] = s * b[i]
- “Add” adds two different elements in memory: a[i] = b[i] + c[i]
- “Triad” adds two elements while scaling one: a[i] = b[i] + s * c[i]

Together, these four numbers should help determine the overall memory bandwidth of the system.

Stream is not very run-time configurable. Generally, configuring parameters involves modifying the code.

A memory performance consistency study

We don’t have a good comparison point for Stream. What we can do is run stream across all of our nodes to make sure we get consistent performance:

We can do this with a one-liner salloc:

```
[lowell@te-master STREAM]$ salloc -N10 -n10 -c16 --tasks-per-node=1 srun
 ↪  --mpi=none -o stream.out ./stream_c.exe
salloc: Granted job allocation 107
salloc: Relinquishing job allocation 107
```

Note that we redirected output to stream.out. We can now compare performance across our nodes:

```
[lowell@te-master STREAM]$ grep Copy stream.out
Copy:           17370.9     0.009223     0.009211     0.009237
Copy:           17605.6     0.009109     0.009088     0.009147
Copy:           17348.0     0.009261     0.009223     0.009279
Copy:           17265.8     0.009291     0.009267     0.009333
Copy:           17617.6     0.009098     0.009082     0.009125
Copy:           17397.0     0.009248     0.009197     0.009286
Copy:           17526.5     0.009145     0.009129     0.009165
Copy:           17357.4     0.009247     0.009218     0.009297
Copy:           17338.6     0.009240     0.009228     0.009253
Copy:           17329.1     0.009269     0.009233     0.009315
[lowell@te-master STREAM]$ grep Scale stream.out
Scale:          17131.0     0.009357     0.009340     0.009383
Scale:          17215.3     0.009325     0.009294     0.009352
Scale:          16812.1     0.009541     0.009517     0.009582
Scale:          17028.4     0.009408     0.009396     0.009430
Scale:          17258.3     0.009293     0.009271     0.009323
Scale:          17254.3     0.009308     0.009273     0.009334
Scale:          17023.2     0.009425     0.009399     0.009445
Scale:          16890.0     0.009507     0.009473     0.009539
Scale:          16884.9     0.009530     0.009476     0.009589
```

```
Scale:          17254.3     0.009299     0.009273     0.009322
[lowell@te-master STREAM]$ grep Add stream.out
Add:            19174.0     0.012545     0.012517     0.012594
Add:            19526.6     0.012318     0.012291     0.012341
Add:            18970.9     0.012675     0.012651     0.012711
Add:            19136.1     0.012573     0.012542     0.012596
Add:            19500.8     0.012333     0.012307     0.012350
Add:            19209.1     0.012528     0.012494     0.012560
Add:            19304.9     0.012454     0.012432     0.012474
Add:            18704.7     0.012854     0.012831     0.012889
Add:            19178.7     0.012555     0.012514     0.012607
Add:            19131.0     0.012574     0.012545     0.012604
[lowell@te-master STREAM]$ grep Triad stream.out
Triad:          19772.8     0.012149     0.012138     0.012174
Triad:          19498.2     0.012334     0.012309     0.012343
Triad:          19663.9     0.012225     0.012205     0.012239
Triad:          19431.6     0.012392     0.012351     0.012431
Triad:          19381.4     0.012399     0.012383     0.012434
Triad:          19646.6     0.012237     0.012216     0.012256
Triad:          19638.6     0.012257     0.012221     0.012283
Triad:          19340.8     0.012431     0.012409     0.012448
Triad:          19550.4     0.012321     0.012276     0.012362
Triad:          19216.8     0.012517     0.012489     0.012546
```

We can see that there is some variance in the results, but nothing that would indicate that any single node performs especially worse than any other.

Try plugging this data into a spreadsheet and computing the =STDEV(). In my case, Add had a good deal more variance than other operations. The 8th entry is especially low. It might be worth re-running to see if that is consistent.

Step 2: Network performance benchmarking

We will look at two different network performance tools. One (Netperf) will measure performance at the TCP layer, the other will measure low-level Infiniband performance.

Netperf TCP benchmarks

Building Netperf & running

Netperf can be found at: https://github.com/HewlettPackard/netperf

It's a straight-forward build:

```
[lowell@te-master ~]$ git clone https://github.com/HewlettPackard/netperf
Cloning into 'netperf'...
remote: Enumerating objects: 4928, done.
remote: Total 4928 (delta 0), reused 0 (delta 0), pack-reused 4928
Receiving objects: 100% (4928/4928), 15.21 MiB | 11.78 MiB/s, done.
Resolving deltas: 100% (3696/3696), done.
[lowell@te-master ~]$ cd netperf
[lowell@te-master netperf]$ ./autogen.sh
configure.ac:28: installing './compile'
[lowell@te-master netperf]$ mkdir build
[lowell@te-master netperf]$ cd build
[lowell@te-master build]$ ../configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
...
```

Note: ./autogen.sh will create the ./configure script for us.

Now build:

```
[lowell@te-master build]$ make -j32
make  all-recursive
make[1]: Entering directory `/home/lowell/netperf/build'
...
make[2]: Leaving directory `/home/lowell/netperf/build'
make[1]: Leaving directory `/home/lowell/netperf/build'
```

We can just use the executables in our build directory without installing them:

```
[lowell@te-master build]$ ls src
Makefile           netlib.o           netserver.o     nettest_sctp.o
dscp.o             netperf            netsh.o         nettest_sdp.o
missing            netperf.o          nettest_bsd.o   nettest_unix.o
net_uuid.o         netperf_version.h  nettest_dlpi.o  nettest_xti.o
netcpu_procstat.o  netserver          nettest_omni.o
```

Netperf works by running netserver on one host, and connecting to it with netperf on another.

Let's open two terminals to two different nodes.

To start the server on the first node:

```
[lowell@te01 src]$ ./netserver -D -4
Starting netserver with host 'IN(6)ADDR_ANY' port '12865' and family AF_INET
```

On the other node we can run our tests:

```
[lowell@te02 src]$ ./netperf -H te01
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to te01 ()
 ↪  port 0 AF_INET
```

```
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.02     941.42
```

By default, with no other options, netperf will perform a bandwidth stream test. As we can see here, I'm getting about 0.941 Gbps over this connection. That's not surprising because this is a gigabit link.

This can be a good troubleshooting tool when we suspect network issues.

Let's run another test, this time focused on capturing latency:

```
[lowell@te02 src]$ ./netperf -H te01 -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
 ↪  te01 () port 0 AF_INET : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec

16384  87380  1        1       10.00    11496.13
```

We can put this in the more usual terms of latency with: 10/11496.13 ∗ 10^6 = 869 ms.

Netperf has quite a few other tests available.

Let's see how the Infiniband performance compares. Keep in mind, this will be the performance of the IPoIB IP emulation layer for Infiniband, not low-level performance:

```
[lowell@te02 src]$ ./netperf -H 192.168.0.1
MIGRATED TCP STREAM TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
 ↪  192.168.0.1 () port 0 AF_INET
Recv   Send    Send
Socket Socket  Message  Elapsed
Size   Size    Size     Time     Throughput
bytes  bytes   bytes    secs.    10^6bits/sec

 87380  16384  16384    10.00    15468.81
[lowell@te02 src]$ ./netperf -H 192.168.0.1 -t TCP_RR
MIGRATED TCP REQUEST/RESPONSE TEST from 0.0.0.0 (0.0.0.0) port 0 AF_INET to
 ↪  192.168.0.1 () port 0 AF_INET : first burst 0
Local /Remote
Socket Size   Request  Resp.   Elapsed  Trans.
Send   Recv   Size     Size    Time     Rate
bytes  Bytes  bytes    bytes   secs.    per sec
```

```
16384  87380  1        1       10.00    46968.78
```

So, we're getting about 15.4 Gbps bandwidth, and 213 ms latency. This is nowhere near advertised IB specs, but clearly much better than the ethernet. The low performance is due to the emulation layer.

Infiniband bandwidth & latency benchmarks

The OFED stack we installed for InfiniBand came with a couple of tools that can help us get these numbers without the emulation layer. Let's use them to compare.

Let's do bandwidth tests first:

On one node we run ib_write_bw with no host specified, on the other ib_write_bw <host>:

```
[root@te01 ~]# ib_write_bw -F

************************************
* Waiting for client to connect... *
************************************
```

On the client side:

```
[root@te02 ~]# ib_write_bw -F 192.168.0.1
---------------------------------------------------------------------------
↪  --
                    RDMA_Write BW Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 TX depth        : 128
 CQ Moderation   : 100
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 0[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------
↪  --
 local address: LID 0x03 QPN 0x3b51 PSN 0x196d8d RKey 0x5bf0b9 VAddr
↪  0x007f1675c70000
 remote address: LID 0x01 QPN 0x680b PSN 0x3d5056 RKey 0x904008 VAddr
↪  0x007f3fc2170000
---------------------------------------------------------------------------
↪  --
```

```
#bytes     #iterations    BW peak[MB/sec]    BW average[MB/sec]
↪  MsgRate[Mpps]
65536      5000             11812.36            11811.81
↪  0.188989
---------------------------------------------------------------------------
↪  --
```

That's about 88 Gbps. The theoretical peak is 25 ∗ 4x Gbps ~= 100 Gbps. Not too bad.

Let's test latency:

```
[root@te02 ~]# ib_write_lat -F 192.168.0.1
---------------------------------------------------------------------------
↪  --
                    RDMA_Write Latency Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 TX depth        : 1
 Mtu             : 4096[B]
 Link type       : IB
 Max inline data : 220[B]
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------
↪  --
 local address: LID 0x03 QPN 0x3b52 PSN 0xedf8 RKey 0x2f95f9 VAddr
↪  0x007f4a191fd000
 remote address: LID 0x01 QPN 0x680c PSN 0x1bd375 RKey 0x2e95ce VAddr
↪  0x007f887710d000
---------------------------------------------------------------------------
↪  --
 #bytes #iterations    t_min[usec]    t_max[usec]  t_typical[usec]
↪  t_avg[usec]    t_stdev[usec]   99% percentile[usec]   99.9%
↪  percentile[usec]
 2       1000          0.79           7.98         0.80                0.81
↪               0.13            0.82       7.98
---------------------------------------------------------------------------
↪  --
```

That's about 0.8 usec latency. Theoretical is 0.61 usec for EDR InfiniBand.

Step 3: MPI benchmarks (IMB)

The Intel MPI Benchmarks measure over-all MPI performance. This is largely a test of the network fabric but also tests MPI library speed.

We can get the IMB software from: https://github.com/intel/mpi-benchmarks

```
[lowell@te-master ~]$ git clone https://github.com/intel/mpi-benchmarks
Cloning into 'mpi-benchmarks'...
...
```

We will only build the C benchmarks, not the C++.

```
[lowell@te-master ~]$ cd mpi-benchmarks/src_c
[lowell@te-master src_c]$ make -j32
...
mpicc -DMPIIO -DIO -DIMB2018 -c IMB_init_transfer.c -o
 ↪  build_IO/IMB_init_transfer.o
mpicc -DMPIIO -DIO -DIMB2018 -c IMB_chk_diff.c -o build_IO/IMB_chk_diff.o
mpicc  build_IO/IMB.o build_IO/IMB_utils.o build_IO/IMB_declare.o
 ↪  build_IO/IMB_init.o build_IO/IMB_mem_manager.o build_IO/IMB_init_file.o
 ↪  build_IO/IMB_user_set_info.o build_IO/IMB_benchlist.o
 ↪  build_IO/IMB_parse_name_io.o build_IO/IMB_strgs.o
 ↪  build_IO/IMB_err_handler.o build_IO/IMB_g_info.o build_IO/IMB_warm_up.o
 ↪  build_IO/IMB_output.o build_IO/IMB_cpu_exploit.o
 ↪  build_IO/IMB_open_close.o build_IO/IMB_write.o build_IO/IMB_read.o
 ↪  build_IO/IMB_init_transfer.o build_IO/IMB_chk_diff.o -o IMB-IO
make[1]: Leaving directory `/home/lowell/mpi-benchmarks/src_c'
```

Once completed, we should submit these jobs using Slurm:

```
[lowell@te-master src_c]$ salloc -N2 -n2 srun --mpi=pmix ./IMB-MPI1
 ↪  AllReduce
...
# Allreduce

#----------------------------------------------------------------
# Benchmarking Allreduce
# #processes = 2
#----------------------------------------------------------------
       #bytes #repetitions  t_min[usec]  t_max[usec]  t_avg[usec]
            0         1000         0.04         0.04         0.04
            4         1000         1.20         1.76         1.48
            8         1000         0.84         2.20         1.52
           16         1000         0.94         2.11         1.52
           32         1000         1.18         1.88         1.53
           64         1000         1.36         1.81         1.58
          128         1000         1.88         2.43         2.15
          256         1000         2.27         2.32         2.30
          512         1000         2.04         2.92         2.48
         1024         1000         2.13         3.49         2.81
         2048         1000         2.59         4.35         3.47
         4096         1000         4.08         6.11         5.09
         8192         1000         6.74         8.44         7.59
        16384         1000        13.21        13.34        13.28
        32768         1000        21.74        22.03        21.88
        65536          640        31.02        32.32        31.67
       131072          320        61.93        63.27        62.60
       262144          160       116.15       117.10       116.63
       524288           80       209.87       211.98       210.93
      1048576           40       391.60       393.57       392.59
      2097152           20       762.43       764.56       763.49
      4194304           10      1574.83      1577.18      1576.00
```

There are many different MPI tests available in IMB, one for each type of MPI operation. We can run a whole set up tests with:

```
[lowell@te-master src_c]$ salloc -N10 -n10 srun --mpi=pmix ./IMB-MPI1
 ↪  -multi 0
...(huge amount of output)
```

As we can see from both of these results, the latency increases a bit in a realistic MPI operation.

A good comparison (*Challenge*) would be to build IMB using Intel MPI instead of OpenMPI and compare the results.

Step 4: HPL benchmarks

Undoubtedly the most ubiquitous benchmark in HPC is the HPL ("High Performance Linpack") benchmark. This is the benchmark that top500.org uses to rank systems. It's long been considered the gold standard for macro system performance, but that has become more debated in recent users, and benchmarks like the HPCG have become popular.

Building HPL

HPL can be a bit tricky to compile. It can be even trickier if you are trying to tweak every bit of performance out of it. You can grab the latest copy of HPL at: https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz.

```
[lowell@te-master ~]$ wget
 ↪  https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz
--2019-06-17 00:51:00--  https://www.netlib.org/benchmark/hpl/hpl-2.3.tar.gz
Resolving www.netlib.org (www.netlib.org)... 160.36.131.221
Connecting to www.netlib.org (www.netlib.org)|160.36.131.221|:443...
 ↪  connected.
HTTP request sent, awaiting response... 200 OK
```

```
Length: 660871 (645K) [application/x-gzip]
Saving to: ‘hpl-2.3.tar.gz.1’

100%[==========================================================================
 ↪  ====>] 660,871     1.46MB/s   in
 ↪  0.4s

2019-06-17 00:51:01 (1.46 MB/s) - ‘hpl-2.3.tar.gz’ saved [660871/660871]
```

Extract it:

```
[lowell@te-master ~]$ tar zxvf hpl-2.3.tar.gz
...
hpl-2.3/www/spreadM.jpg
hpl-2.3/www/tuning.html
```

I have provided a Make.intel64 file in /data/ on te−cm. You will need to place this file in the root of the hpl−2.3 directory to build HPL.

Once the file is in place, we need to make sure that openblas is loaded in our lmod environment:

```
[lowell@te-master hpl-2.3]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc

[lowell@te-master hpl-2.3]$ module load openblas
[lowell@te-master hpl-2.3]$ module list
Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
 ↪   6) openblas/0.3.5
```

Now we can build HPL with:

```
[lowell@te-master hpl-2.3]$ make arch=intel64
...
make[3]: Leaving directory `/home/lowell/hpl-2.3/testing/ptest/intel64'
touch dexe.grd
make[2]: Leaving directory `/home/lowell/hpl-2.3/testing/ptest/intel64'
make[1]: Leaving directory `/home/lowell/hpl-2.3'
```

Note: using the −j <num> option with make seems to break the HPL build process.

If anything goes wrong with the build, you’ll want to clean with:

```
[lowell@te-master hpl-2.3]$ make arch=intel64 clean_arch_all
...
```

Once HPL is built, you can find it under bin/intel64:

```
[lowell@te-master hpl-2.3]$ cd bin/intel64/
[lowell@te-master intel64]$ ls
HPL.dat  xhpl
```

The executable is xhpl. It is an MPI executable.

The file HPL.dat is where you put configuration. The provided version will run a simple 4-processor example. Let's run it to make sure things are working:

```
[lowell@te-master intel64]$ mpirun -np 4 ./xhpl
...(dumps a huge amount of output)
```

The summary lines for each run start with WR<xxxx>. Let's get something more meaningful by grepping for that:

```
[lowell@te-master intel64]$ mpirun -np 4 ./xhpl | grep -E '^WR'
...
WR00C2L4           35     4     4     1                0.00
 ↪  4.9069e-02
WR00C2C2           35     4     4     1                0.00
 ↪  3.7408e-02
WR00C2C4           35     4     4     1                0.00
 ↪  4.5927e-02
WR00C2R2           35     4     4     1                0.00
 ↪  4.5384e-02
WR00C2R4           35     4     4     1                0.00
 ↪  3.5792e-02
WR00R2L2           35     4     4     1                0.00
 ↪  3.5779e-02
WR00R2L4           35     4     4     1                0.00
 ↪  4.3305e-02
WR00R2C2           35     4     4     1                0.00
 ↪  4.8408e-02
WR00R2C4           35     4     4     1                0.00
 ↪  4.8844e-02
WR00R2R2           35     4     4     1                0.00
 ↪  4.7717e-02
WR00R2R4           35     4     4     1                0.00
 ↪  3.9889e-02
...
```

The last column here is a measurement of GFlops. We don't expect high numbers here, because we're running on only 4 cores.

The `HPL.dat` file

Take a look in the HPL.dat file:

```
HPLinpack benchmark input file
Innovative Computing Laboratory, University of Tennessee
HPL.out      output file name (if any)
6            device out (6=stdout,7=stderr,file)
4            # of problems sizes (N)
29 30 34 35  Ns
4            # of NBs
1 2 3 4      NBs
0            PMAP process mapping (0=Row-,1=Column-major)
3            # of process grids (P x Q)
2 1 4        Ps
2 4 1        Qs
16.0         threshold
3            # of panel fact
0 1 2        PFACTs (0=left, 1=Crout, 2=Right)
2            # of recursive stopping criterium
2 4          NBMINs (>= 1)
1            # of panels in recursion
2            NDIVs
3            # of recursive panel fact.
0 1 2        RFACTs (0=left, 1=Crout, 2=Right)
1            # of broadcast
0            BCASTs (0=1rg,1=1rM,2=2rg,3=2rM,4=Lng,5=LnM)
1            # of lookahead depth
0            DEPTHs (>=0)
2            SWAP (0=bin-exch,1=long,2=mix)
64           swapping threshold
0            L1 in (0=transposed,1=no-transposed) form
0            U  in (0=transposed,1=no-transposed) form
1            Equilibration (0=no,1=yes)
8            memory alignment in double (> 0)
```

There are a lot of parameters in here, but only a few that we care about.

We won't use any of the output file parameters. In the end, we'll let Slurm take care of outputting results to a file for us.

The N field ("# of problem sizes") specifies how many of the following "NBs" we will be giving. In this case, we have 4. All of the # fields work like this, specifying how many of the next field type will be specified.

The Ns specifies the problem size. This ultimately controls how big the matrix we're going to work with is.

The next part that especially matters is the "process grids". This says how HPL will split up the problem across nodes and cores. So, if you want to run across a total of 10 nodes and 32 cores, that's 320 cores, and the process grid should multiply out to 320. The general rule of thumb is that you want to be as close to "square" as you can, i.e. P close to Q. So, for 320, 16 x 20 is likely the best choice. Usually, you want P < Q.

HPL has been studied for many years, and a lot of patterns have been found. If you go to https://www.advancedclustering.com/act_kb/tune-hpl-dat-file/ there is a reasonably good calculator for estimating the best HPL.dat file.

Another good resource is: http://hpl-calculator.sourceforge.net. This calculator doesn't provide you with the HPL.dat, but it will tell you how close you get to the theoretical max.

Making a Slurm job for HPL

The real HPL jobs will run for a little while. We'll want to submit them as batch jobs. Let's put together an example batch job and try it out. This will also give us more experience with sbatch scripts, especially with mpi.

The first thing to know is that srun is MPI aware. When we tested out our build of HPL, we ran it inside of the mpirun command and gave it a number of processes. This is the traditional way to start an MPI process, along with a hostlist file that would indicate which hosts to use. This is how you would start an MPI job if we didn't have a WLM system.

srun handles all of this bring-up for us. We don't even have to say which hosts to run on, or how many processes; it will take care of all of that.

We *do* have to think carefully about how we want to arrange our processes. HPL uses MPI for *all* of its communication, unlike some programs that combine MPI with multi-threading for instance. This means that Slurm needs to start an HPL process for each core we want to run on.

To get started, we'll make a 2-node job. That will be 2-nodes, and 64 tasks. Make a directory called 2−node under bin/intel64 under the HPL directory. We will run in this directory:

```
[lowell@te-master ~]$ cd hpl-2.3/bin/intel64/
[lowell@te-master intel64]$ ls
HPL.dat  xhpl
[lowell@te-master intel64]$ mkdir 2-node
[lowell@te-master intel64]$ cd 2-node/
```

First, create a 2-node HPL.dat file. You can use the calculator from above. The system has 32 cores, 2 nodes, and 64GB RAM. You can leave the block size, NB, as the default. Write the result to HPL.dat in the 2−node directory.

Now, create a script called hpl.sbatch. Given it the following contents:

```
#!/bin/bash
#SBATCH --job-name=hpl-32x2
#SBATCH --ntasks=64
#SBATCH --nodes=2
#SBATCH --output=hpl_%j.log
```

```
pwd; hostname; date

module load openmpi3 openblas

srun --mpi=pmix ../xhpl

date
```

The sbatch parameters we gave provide a unique job name, that we need 64 tasks across 2 nodes, and say that the output should go to hpl_<jobid>.log, where <jobid> is the numerical job id that slurm assigns when the job runs. Using the %j keeps us from over-writing the output file if we run multiple times.

The next line, pwd; hostname; date are just to provide some convenient bookkeeping for possible later troubleshooting.

The real command is srun −−mpi=pmix ../xhpl. xhpl will use the HPL.dat that sits in the current working directory. The −−mpi=pmix option tells Slurm to use pmix for process management (there are various options, but this is the one we're set up for).

We can now sbatch our job:

```
[lowell@te-master 2-node]$ sbatch hpl.sbatch
Submitted batch job 115
[lowell@te-master 2-node]$ squeue
             JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
               115   cluster hpl-32x2   lowell  R       0:03      2
 ↪  te[01-02]
[lowell@te-master 2-node]$ ls
hpl_115.log  HPL.dat  hpl.sbatch
```

Our job is running. We can tail its output:

```
[lowell@te-master 2-node]$ tail hpl_115.log

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be
 ↪  1.110223e-16
- Computational tests pass if scaled residuals are less than
 ↪  16.0

Column=000000192 Fraction= 0.2% Gflops=9.465e+03
Column=000000384 Fraction= 0.3% Gflops=1.402e+03
Column=000000576 Fraction= 0.5% Gflops=1.085e+03
```

This will take a while to run.

An HPL scaling study

Now that 2−node is running, make a similar directory for 1−node, 4−node, 8−node, and 10−node.

If you use the calculator for the 10−node it will give you an improper config. You can use what it gives you, but change the Ns line to be the same is what you have for 8−node.

Once those are all set up, sbatch them.

I started all of mine at once like this:

```
[lowell@te-master intel64]$ for i in 1 2 4 8 10; do echo $i; ( cd $i-node
 ↪  ; sbatch hpl.sbatch ) ; done
1
Submitted batch job 116
2
Submitted batch job 117
4
Submitted batch job 118
8
Submitted batch job 119
10
Submitted batch job 120
[lowell@te-master intel64]$ squeue
            JOBID PARTITION     NAME     USER ST       TIME  NODES
 ↪  NODELIST(REASON)
              119   cluster hpl-32x8   lowell PD       0:00      8
 ↪  (Resources)
              120   cluster hpl-32x1   lowell PD       0:00     10
 ↪  (Priority)
              116   cluster hpl-32x1   lowell  R       0:02      1 te01
              117   cluster hpl-32x2   lowell  R       0:02      2
 ↪  te[02-03]
              118   cluster hpl-32x4   lowell  R       0:02      4
 ↪  te[04-07]
```

At this point, you should verify that all of the jobs are starting without error by checking the hpl_∗.log files.

This will take a while, and we will likely work on something else for a bit while it runs.

Once it has finished, get all of the final values. We're going to want to plot these to see if we are scaling linearly. We can collect the values with:

```
[lowell@te-master intel64]$ grep WR */hpl_*.log
10-node/hpl_103.log:WR11C2R4      234240   192    16    20
↪  2462.54             3.4795e+03
1-node/hpl_93.log:WR11C2R4       82560   192     4     8             1021.66
↪              3.6722e+02
2-node/hpl_95.log:WR11C2R4      117120   192     8     8             1469.01
↪              7.2910e+02
4-node/hpl_96.log:WR11C2R4      165504   192     8    16             2094.34
↪              1.4431e+03
8-node/hpl_97.log:WR11C2R4      234240   192    16    16             3038.27
↪              2.8201e+03
```

Now, put these values into Microsoft Excel. Make an X-Y scatter plot (will demo how to do this live). Do you get linear scaling? Are 10 nodes 10x as fast as 1?

Parallel Programming with Threads

Overview

This guide will attempt to introduce parallel programming concepts with little assumption of prior experience. We will not be writing code directly, but we will go through a series of examples to illustrate the process. We will be using the simple non-interacting gas simulator as the background project through this and the next guide. All examples will be in Python 3. The gas.py simulator uses language features that require Python >= 3.7.

Most of the steps in this guide can be followed through individually as non-root users.

Step 1: Setting up the Python environment & introducing `gas.py`

We will be using the gas.py non-interacting gas simulator as an example through this and the next guide. We will need a more modern version of python (>=3.7). We will use pyenv to set this up. First, we need to install pyenv itself. As your user run:

```
[lowell@te-master ~]$ curl -L https://raw.githubusercontent.com/yyuu/pyenv⌋
↪  -installer/master/bin/pyenv-installer |
↪  bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time
↪  Current
                                 Dload  Upload   Total   Spent    Left
↪  Speed
100  2446  100  2446    0     0   5418      0 --:--:-- --:--:-- --:--:--
↪  5423
Cloning into '/home/lowell/.pyenv'...
```

```
...

WARNING: seems you still have not added 'pyenv' to the load path.

# Load pyenv automatically by adding
# the following to ~/.bashrc:

export PATH="/home/lowell/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Open your $HOME/.bash_profile and add towards the bottom (as suggested by the installer):

```
export PATH="/home/lowell/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"

export PYENV_VIRTUALENV_DISABLE_PROMPT=1
```

We can activate this change right away by “sourcing” our profile:

```
[lowell@te-master ~]$ source .bash_profile
```

We should now have the pyenv command available.

```
[lowell@te-master ~]$ pyenv help
Usage: pyenv <command> [<args>]

Some useful pyenv commands are:
   commands    List all available pyenv commands
   local       Set or show the local application-specific Python version
   global      Set or show the global Python version
   shell       Set or show the shell-specific Python version
   install     Install a Python version using python-build
   uninstall   Uninstall a specific Python version
   rehash      Rehash pyenv shims (run this after installing executables)
   version     Show the current Python version and its origin
   versions    List all Python versions available to pyenv
   which       Display the full path to an executable
   whence      List all Python versions that contain the given executable

See `pyenv help <command>' for information on a specific command.
For the full documentation, see: https://github.com/pyenv/pyenv#readme
```

Pyenv allows us to build and maintain multiple version of Python and switch between them. It can also set new global default python installs, as well as set that we should always use a particular python version in a particular directory. We can see what versions it can install with:

```
[lowell@te-master ~]$ pyenv install -l
Available versions:
  2.1.3
  2.2.3
  2.3.7
  2.4.0
  2.4.1
...
```

The one we will use is anaconda3−2019.03. Anaconda is a bundled version of python that comes with various scientific software available. Let's install it (this will take a few minutes):

```
[lowell@te-master ~]$ pyenv install anaconda3-2019.03
Downloading Anaconda3-2019.03-Linux-x86_64.sh...
-> https://repo.continuum.io/archive/Anaconda3-2019.03-Linux-x86_64.sh
Installing Anaconda3-2019.03-Linux-x86_64...
Installed Anaconda3-2019.03-Linux-x86_64 to
 ↪  /home/lowell/.pyenv/versions/anaconda3-2019.03
```

We can check what's available with:

```
[lowell@te-master ~]$ pyenv versions
* system (set by /home/lowell/.pyenv/version)
  anaconda3-2019.03
```

Let's globally select anaconda3−2019.03 as our version:

```
[lowell@te-master ~]$ python --version
Python 2.7.5
[lowell@te-master ~]$ pyenv global anaconda3-2019.03
[lowell@te-master ~]$ python --version
Python 3.7.3
```

If you ever want to switch back to the system provided version, use:

```
[lowell@te-master ~]$ pyenv global system
[lowell@te-master ~]$ python --version
Python 2.7.5
```

With anaconda3−2019.03 enabled, let's move ahead with testing our gas.py simulator.

You can find gas.py in the bundle /data/gas.tar.gz on the CM. Extract this in your home directory, then go into the gas directory.

```
[lowell@te-master ~]$ cd gas
[lowell@te-master gas]$ ls
Box.py  Cell.py  gas.py  Particle.py  split.py  World.py
```

The only file we will be dealing with is gas.py. This is where the main logic of the simulator is located and is the script we will execute. Briefly, the other files do the following:

- Box.py contains some basic dataclasses that gas.py uses
- Cell.py contains the logic surrounding Cells and Cell boundary conditions
- Particle.py contains particle logic, including most of the physics
- split.py is a utility we will make use of later that splits gas.py output into multiple files for easier use with ParaView (visualization tool)
- World.py contains a class that defines the global characteristics of our simulation

We can see what options gas.py provides:

```
[lowell@te-master gas]$ ./gas.py -h
Usage: ./gas.py
   -h, --help       : Print this usage info
   -p, --part <#>   : Number of particles (default: 10)
   -s, --steps <#>  : Number of timesteps (default: 1)
   -o, --out <file> : Output to file (default: stdout)
```

We can specify the number of particles (−p), a number of timesteps (−s), and an output file for generated data (−o).

Gas works like most physics simulators. It divides the simulation into "timesteps". At each timestep, it moves the particles for a small period of time. It then computes if any exceptions like collisions have occurred, and deals with them. Finally, it figures out if any particles have crossed boundaries, and computes their boundary conditions.

For simplicity sake, you may notice that there is no way to feed initial conditions into Gas. Gas always generates a random particle distribution every time it runs.

Most particle simulators use a similar approach. In general, particle simulators would use force interactions. In this case, at each timestep, we first compute all of the forces and derive accelerations from them, then we "integrate" the timestep. The same basic concepts are used.

We can make sure Gas is working by running once with the default options:

```
[lowell@te-master gas]$ ./gas.py
0.0009 running with 10 particles for 1 steps, output to <stdout>
--- ts=0
0.01393473347246452,0.0043420756285396335,2.8700354969971222,8.650786462548
 ↪  004,8.902126505175913,0.024024720114272098,0.022034620418221525,0.01711
 ↪  409688452525
...
0.0014 timestep=1, particles=10
  0.0014 comb particles into cells
    Cell 0: 0 particles added to our cell
  0.0015 start integrate
  0.0015 start collide
    Cell 0: resolved 0 collisions
  0.0019 start boundary check
    Cell 0: 0 particles out-of-bounds
  0.0020 resolve out-of-bounds particles
  reflected 0 particles of 0 in the oob list
  0.0020 start print
--- ts=1
0.013934733347246452,0.0043420756285396335,2.871236733002836,8.6518881935689
↪  15,8.90298221002014,0.024024720114272098,0.022034620418221525,0.0171140
↪  9688452525
...
0.0022 done
```

This is a *very* boring simulation. It only computed one timestep, and nothing really happened. But, it worked.

Let's get a little familiarity with the structure of the code. Inside gas.py, we find a class called Sym. This is where everything we will care about happens. The run() method of this class is the main simulation loop. It computes the following steps:'

1. "comb" the set of particles to see what belongs in our Cell (box). This step is a little superfluous now but will prove handy later.
2. "integrate" the timestep, i.e. move particles for a brief instant
3. figure out which particles "collide"; resolve those collisions (i.e. figure out how they scattered off of each other. This bit is handled by the class in Particle.py)
4. check to see if any particles went outside of the cell boundaries, make a list of them
5. apply boundary conditions to the particles that left the cell. By default, our boundary conditions are "hard", i.e. the particles bounce off the boundary.

Take a moment to look through this class to see how it works. We'll be making changes to this as we go along.

```python
# Simulation loop

class Sym:
    def boundary_resolve(self):
        c = 0
        for p in self.w.oob:
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
         ↪  len(self.w.oob)), file=sys.stderr)

    def run(self):
        c = Cell(0, self.w, parts, self.w.box)
        print("--- ts=0", file=out)
        c.print(out)
        for i in range(1,steps+1):
            print("{:0.4f} timestep={},
             ↪  particles={}".format(since(self.w), i, len(c.parts)),
             ↪  file=sys.stderr)
            print("\t{:0.4f} comb particles into
             ↪  cells".format(since(self.w)), file=sys.stderr)
            c.comb_particles()
            print("\t{:0.4f} start integrate".format(since(self.w)),
             ↪  file=sys.stderr)
            c.integrate()
            print("\t{:0.4f} start collide".format(since(self.w)),
             ↪  file=sys.stderr)
            c.collide()
            print("\t{:0.4f} start boundary check".format(since(self.w)),
             ↪  file=sys.stderr)
            c.boundary_check()
            print("\t{:0.4f} resolve out-of-bounds
             ↪  particles".format(since(self.w)), file=sys.stderr)
            self.boundary_resolve()
            print("\t{:0.4f} start print".format(since(self.w)),
             ↪  file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            c.print(out)

    def __init__(self, w: World):
        self.w = w
```

Step 2: Visualizing our simulation

We're going to go ahead and run a simple example simulation and visualize it. This will help us check ourselves as we work along. We won't repeat these steps beyond this point, but you can always go back and check with any new simulation we run.

We are going to use ParaView for visualization. Go ahead and download it and install it on your laptop now.

Let's run a simulation. We will keep it small so it runs quickly. We will run with 100 particles for 1000 timesteps. We will output to sim.csv. We will redirect stderr to sim.log to keep the flood of messages down:

```
[lowell@te-master gas]$ ./gas.py -p 100 -s 1000 -o sim.csv 2> sim.log
```

This should complete in around 30 seconds.

If you look at the sim.csv file it's full of lines of comma separated numbers and the occasional −−− ts=# line. ParaView, that we are going to be using for visualization will work better if this file is split into one file per timestep instead. This is what the split .py script is for.

```
[lowell@te-master gas]$ ./split.py
usage: ./split.py <file> <prefix>
```

Let's make a directory called sim for these files to go into:

```
[lowell@te-master sim]$ ../split.py ../sim.csv sim
created 1001 timestep files
[lowell@te-master sim]$ ls
sim.1000.csv  sim.1.csv    sim.300.csv
...(LOTS of files)
```

This is a format that ParaView can read in directly. Create a tarball of this directory and move it to your laptop:

```
[lowell@te-master gas]$ tar zcvf sim.tar.gz sim
sim/
sim/sim.1.csv
sim/sim.2.csv
sim/sim.3.csv
sim/sim.4.csv
sim/sim.5.csv
...
sim/sim.1001.csv
```

For the rest of this section we will work through setting up the ParaView visualization interactively. Let the instructor know when you've gotten here.

Step 3: Simple threading examples

Single-threaded

In this section, we'll play with a couple of simple code examples illustrating threading concepts. Make sure you create and run these scripts as we go along, and study their structure.

Let's start with a simple, non-threaded script. This script adds all of the numbers between 1 and 100 (sound familiar?).

```
#!/usr/bin/env python3
# sum.py

sum=0

for i in range(1,101):
  sum += i
  print("{} : {}".format(i, sum))
```

Running this should produce:

```
[lowell@te-master threading-examples]$ ./sum.py
1 : 1
2 : 3
```

```
3 : 6
4 : 10
...
100 : 5050
```

Two threads

We can divide this problem in half with threads. We'll have one thread add the numbers between 1 and 50, and another the numbers between 51 and 100, then add them together in the end.

The general structure for a threaded program is:

1. create the thread and bind it to a function
2. start the threads
3. "join" the threads (i.e. wait for them to stop)
4. finish up

Here's an example that does this:

```
#!/usr/bin/env python3
# sum-2x.py

import threading

low=0
high=0

def sum_low():
  global low
  for i in range(1,51):
    low += i
    print("{} : {}".format(i, low))

def sum_high():
  global high
  for i in range(51,101):
    high += i
    print("{} : {}".format(i, low))

t_low = threading.Thread(target=sum_low)
t_high = threading.Thread(target=sum_high)

t_low.start()
t_high.start()
```

```
t_low.join()
t_high.join()

print("total : {}".format(low + high))
```

Running:

```
[lowell@te-master threading-examples]$ ./sum-2x.py
1 : 1
2 : 3
3 : 6
4 : 10
...
total : 5050
```

One interesting thing already is that sum−2x.py performs slightly worse than sum.py:

```
[lowell@te-master threading-examples]$ time ./sum.py >/dev/null

real    0m0.219s
user    0m0.098s
sys    0m0.133s
[lowell@te-master threading-examples]$ time ./sum-2x.py >/dev/null

real    0m0.235s
user    0m0.113s
sys    0m0.135s
```

This happens because initiating a thread costs more than the win we get from parallelism.

n-threads

Despite the performance loss, let's take this a step further. Let's make a version that takes an argument that specifies how many threads to run.

```
#!/usr/bin/env python3
# sum-n.py

import sys
import threading

s = list()
def sum(id, threads):
  global s
  r = int(100/threads)
  for i in range(r * id + 1 ,r * (id+1) + 1):
    s[id] += i
    print("{} : {}".format(i, s[id]))

# entry
if len(sys.argv) != 2:
  print("Usage: {} <num_threads>".format(sys.argv[0]))
  sys.exit()

threads = int(sys.argv[1])
print("Running with {} threads".format(threads))

t = list()
for i in range(0,threads):
  t.append(threading.Thread(target=sum, args=(i, threads,)))
  s.append(0)

for i in range(0,threads):
  t[i].start()

total = 0
for i in range(0,threads):
  t[i].join()
  total += s[i]

print("total : {}".format(total))
```

This one is a bit more complicated, but it adds a couple of core ideas:

1. the model of making a list of threads and starting/joining them is a common one
2. we can pass arguments to our thread as well
3. we avoided colliding in our sums by storing every thread's results in a unique element of a list

Let's run this code, say, with 2 threads:

```
[lowell@te-master threading-examples]$ ./sum-n.py 2
Running with 2 threads
1 : 1
2 : 3
3 : 6
...
total : 5050
```

Looks good. Let's try 6 threads:

```
[lowell@te-master threading-examples]$ ./sum-n.py 6
Running with 6 threads
1 : 1
2 : 3
```

```
3 : 6
...
96 : 1416
total : 4656
```

Wait, we got a different answer. Why? Think of how we constructed our ranges.

Let's do a quick scaling study:

```
[lowell@te-master threading-examples]$ for i in $(seq 1 20); do time
 ↪  ./sum-n.py $i >/dev/null; done 2>&1 | grep real
real    0m0.233s
real    0m0.242s
real    0m0.246s
real    0m0.243s
real    0m0.225s
real    0m0.225s
real    0m0.229s
real    0m0.230s
real    0m0.236s
real    0m0.229s
real    0m0.236s
real    0m0.239s
real    0m0.231s
real    0m0.226s
real    0m0.228s
real    0m0.206s
real    0m0.224s
real    0m0.228s
real    0m0.231s
real    0m0.233s
```

The threading isn't doing us any favors in this example. In general, threading only is an advantage if the time it takes to make the threads and load the libraries is much smaller than the time each thread spends running. Also, as a side note, Python threading just isn't very good.

Locking

In the example above, we kept track of the sums by putting them all in an item-per-thread array (s) and summing them at the end. If we instead using *locking*, we can actually let the threads to all of the summing and have only one global sum.

```
#!/usr/bin/env python3
# sum-n-lock.py
```

```
import sys
import threading


lock = threading.Lock()
total = 0

def sum(id, threads):
  global total
  r = int(100/threads)
  sum = 0
  for i in range(r * id + 1 ,r * (id+1) + 1):
    sum += i
    print("{} : {}".format(i, sum))
  lock.acquire()
  total += sum
  lock.release()

# entry
if len(sys.argv) != 2:
  print("Usage: {} <num_threads>".format(sys.argv[0]))
  sys.exit()

threads = int(sys.argv[1])
print("Running with {} threads".format(threads))

t = list()
for i in range(0,threads):
  t.append(threading.Thread(target=sum, args=(i, threads,)))

for i in range(0,threads):
  t[i].start()

for i in range(0,threads):
  t[i].join()

print("total : {}".format(total))
```

This is an example of a mutex lock strategy, though Python calls it acquire/release rather than lock/unlock. Specifically, look at these lines:

```
  lock.acquire()
  total += sum
  lock.release(
```

We know that the total variable is safe because it only gets touched when the lock is held, so only one thread at a time is allowed to write to it.

Step 4: Failure modes for parallel codes

Let's look at two quick examples that illustrate race conditions and deadlocks.

Race condition

We're going to intentionally modify sum−2x.py to have a race condition. It's not that hard to do. All we have to do is wait for only *one* of the threads to join:

```
#!/usr/bin/env python3
# sum-race.py

import threading

low=0
high=0

def sum_low():
  global low
  for i in range(1,51):
    low += i
    print("{} : {}".format(i, low))

def sum_high():
  global high
  for i in range(51,101):
    high += i
    print("{} : {}".format(i, low))

t_low = threading.Thread(target=sum_low)
t_high = threading.Thread(target=sum_high)

t_low.start()
t_high.start()

t_low.join()

print("total : {}".format(low + high))
```

Let's try running a couple of times. The first time I ran it I got:

```
[lowell@te-master threading-examples]$ ./sum-race.py
1 : 1
```

```
2 : 3
3 : 6
...
50 : 1275
total : 5050
```

That looks fine. Then I ran it again:

```
[lowell@te-master threading-examples]$ ./sum-race.py
1 : 1
2 : 3
3 : 6
...(middle of the file)
50 : 1275
total : 1653
57 : 861
...
99 : 1275
100 : 1275
```

That doesn't look good at all. What's happening here?

case 1) By not waiting for the high thread to join, by random chance, we might have the situation that high completed before low. In this case, everything is *fine* because high has already finished.

case 2) However, we also have the chance that low finishes before high. In this unfortunate case, we go on to add the total and try to clean up before high has finished its work.

In other words, we have a *race* between high and low. It's up to chance whether or not our code runs as expected.

Race conditions can be notoriously difficult to track down. They almost always result from a failure to *synchronize* our threads/processes appropriately, but then go on to assume that they were synchronized. In our case, our missed synchronization is with the joins. In the final version of gas.py (next guide), we'll see another form called a barrier. A barrier is a call that all threads must make. All threads will wait until the last thread as called barrier and then they will move on.

Deadlock

We can use the sum−n−lock.py to illustrate a deadlock:

```
#!/usr/bin/env python3
# sum-n-deadlock.py

import sys
import threading
```

```
lock = threading.Lock()
total = 0

def add_to_total(i):
  global total
  lock.acquire()
  total += i
  lock.release()

def sum(id, threads):
  r = int(100/threads)
  sum = 0
  for i in range(r * id + 1 ,r * (id+1) + 1):
    sum += i
    print("{} : {}".format(i, sum))
  lock.acquire()
  add_to_total(sum)
  lock.release()

# entry
if len(sys.argv) != 2:
  print("Usage: {} <num_threads>".format(sys.argv[0]))
  sys.exit()

threads = int(sys.argv[1])
print("Running with {} threads".format(threads))

t = list()
for i in range(0,threads):
  t.append(threading.Thread(target=sum, args=(i, threads,)))

for i in range(0,threads):
  t[i].start()

for i in range(0,threads):
  t[i].join()

print("total : {}".format(total))
```

We've modified our code to offload the summing of the total to the function add_to_total. In principle, this can be a good idea (more on this in a minute). Let's see what happens when we run it:

```
[lowell@te-master threading-examples]$ ./sum-n-deadlock.py 2
```

```
Running with 2 threads
1 : 1
2 : 3
...
50 : 1275
^CTraceback (most recent call last):
  File "./sum-n-deadlock.py", line 43, in <module>
    t[i].join()
  File "/home/lowell/.pyenv/versions/anaconda3-2019.03/lib/python3.7/thread⌟
 ↪  ing.py", line 1032, in
 ↪  join
    self._wait_for_tstate_lock()
  File "/home/lowell/.pyenv/versions/anaconda3-2019.03/lib/python3.7/thread⌟
 ↪  ing.py", line 1048, in
 ↪  _wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
^CException ignored in: <module 'threading' from '/home/lowell/.pyenv/versi⌟
 ↪  ons/anaconda3-2019.03/lib/python3.7/threading.py'>
Traceback (most recent call last):
  File "/home/lowell/.pyenv/versions/anaconda3-2019.03/lib/python3.7/thread⌟
 ↪  ing.py", line 1281, in
 ↪  _shutdown
    t.join()
  File "/home/lowell/.pyenv/versions/anaconda3-2019.03/lib/python3.7/thread⌟
 ↪  ing.py", line 1032, in
 ↪  join
    self._wait_for_tstate_lock()
  File "/home/lowell/.pyenv/versions/anaconda3-2019.03/lib/python3.7/thread⌟
 ↪  ing.py", line 1048, in
 ↪  _wait_for_tstate_lock
    elif lock.acquire(block, timeout):
KeyboardInterrupt
```

It hung forever, and I had to ^C to kill it. What went wrong?

Well, we got the lock, then we called add_to_total, which in turn asked for the lock… but we already have the lock. So, it deadlocked. It would hang like this forever waiting for the lock to clear because of the circular locking dependency.

If we simply remove the locking statements from the sum method and keep them in the add_to_total method, this can be a good way to actually avoid deadlocks. The idea is: 1) to move the locking close to the operation that needs to be locked; 2) restrict access to shared variables to functions that do locking for you. This makes the operations behave almost like atomic operations (i.e. cannot be interrupted).

Step 5: Multi-threaded gas.py

Now that we've seen some basics of how threads work in python, let's check out the threaded version of Gas. The threaded version is in a different branch. We can just check it out:

```
[lowell@te-master gas]$ git checkout threaded
Switched to branch 'threaded'
```

Let's look at the updated Sym object:

```
class Sym(threading.Thread):
    def boundary_resolve(self):
        c = 0
        for p in self.w.oob:
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
         ↪  len(self.w.oob)), file=sys.stderr)

    def run_cell(self, c):
        print("\t{:0.4f} comb particles into cells".format(since(self.w)),
         ↪  file=sys.stderr)
        c.comb_particles()
        print("\t{:0.4f} start integrate".format(since(self.w)),
         ↪  file=sys.stderr)
        c.integrate()
        print("\t{:0.4f} start collide".format(since(self.w)),
         ↪  file=sys.stderr)
        c.collide()
        print("\t{:0.4f} start boundary check".format(since(self.w)),
         ↪  file=sys.stderr)
        c.boundary_check()

    def print(self):
        for c in self.cells:
            c.print(self.w.out)

    def run(self):
        for i in range(0,threads):
            zrange = (self.w.box.z.high - self.w.box.z.low)/threads
            c = Cell(i, self.w, int(parts/threads), Box(self.w.box.x,
             ↪  self.w.box.y, Range(i*zrange, (i+1)*zrange)))
            self.cells.append(c)

        # create a cell
        print("Running with {} threads".format(threads))
        print("{:0.4f} running with {} particles for {} steps, output to
         ↪  {}".format(since(self.w), parts, steps, out.name),
         ↪  file=sys.stderr)

        print("--- ts=0", file=out)
        self.print()

        for i in range(1, steps+1):
            tds = list()
            for c in self.cells:
                print("{:0.4f} timestep={},
                 ↪  particles={}".format(since(self.w), i, len(c.parts)),
                 ↪  file=sys.stderr)
                tds.append(threading.Thread(target=self.run_cell,
                 ↪  args=(c,)))

            for td in tds:
                print("starting thread")
                td.start()

            for td in tds:
                td.join()

            print("\t{:0.4f} resolve out-of-bounds
             ↪  particles".format(since(self.w)), file=sys.stderr)
            self.boundary_resolve()
            print("\t{:0.4f} start print".format(since(self.w)),
             ↪  file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            self.print()

        print("{:0.4f} done".format(since(self.w)), file=sys.stderr)
        out.close()


    def __init__(self):
        self.cells = list()
        self.w = World(steps=steps, out=out, parts=parts, cells=cells)
```

There are quite a few changes here. This is how it works:

1. for each thread we create a cell
2. each cell gets a slice of the box, where we make threads slices along the z-axis
3. threads are managed and launched much like our sum−n.py, in arrays
4. individual cells are combed, integrated, collided and boundary detected in their threads. When particles leave a cell they're added to an oob (out-of-bounds) list

5. after all threads rejoin, the main process works out boundary resolution for any oob particles that are out-of-bounds for the whole simulation and continues to the next loop iteration

Like sum−n.py, we can choose an arbitrary number of threads. Note that in this version, the “combing” operation is actually necessary. That’s how a cell figures out if any of the oob particles entered its region. If they did, “comb” claims them and removes them from the oob list.

Note that we now have a −t option to specify the number of threads:

```
[lowell@te-master gas]$ ./gas.py -h
Usage: ./gas.py
    -h, --help       : Print this usage info
    -p, --part <#>   : Number of particles (default: 10)
    -s, --steps <#>  : Number of timesteps (default: 1)
    -o, --out <file> : Output to file (default: stdout)
    -t, --threads <file> : Number of threads (default: 1)
```

Let’s do a quick scaling study with 1, 2, and 4 threads.

```
[lowell@te-master gas]$ for i in 1 2 4; do echo $i; time ./gas.py -t$i
 ↪  -p2048 -s 4 >/dev/null 2>&1; done
1

real    1m10.642s
user    1m10.482s
sys    0m0.171s
2

real    1m25.563s
user    1m13.799s
sys    0m46.325s
4

real    1m19.427s
user    0m57.309s
sys    1m13.709s
```

Why are these performing so badly? It’s actually taking longer to run more threads!

Let’s use the strace command to investigate. strace is an extremely handy command. Generally, it gives a listing of every syscall a command run underneath it makes. Try this:

```
[lowell@te-master gas]$ strace ls
execve("/usr/bin/ls", ["ls"], [/* 71 vars */]) = 0
brk(NULL)                               = 0x18c3000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
 ↪  0x7f3e0fb79000
```

```
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or
 ↪  directory)
open("/opt/ohpc/pub/mpi/openmpi3-gnu8/3.1.3/lib/tls/x86_64/libselinux.so.1"⌋
 ↪  , O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or
 ↪  directory)

...(lots more lines)

fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
 ↪  0x7f3e0fb78000
write(1, "Box.py\tCell.py  gas.py  Particle"..., 69Box.py    Cell.py
 ↪  gas.py  Particle.py  __pycache__  split.py  World.py
) = 69
close(1)                                = 0
munmap(0x7f3e0fb78000, 4096)            = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

This can be a very hand way to see why a program is failing, for instance. E.g., did it try to open a file it couldn't open?

strace also has a mode that is like a light-weight profiler for syscalls. If you run it with −c it will give you statistics on where it spent its time in syscalls:

```
[lowell@te-master gas]$ strace -c ls
Box.py    Cell.py  gas.py  Particle.py  __pycache__  split.py  World.py
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 19.94    0.000254           9        29           mmap
 19.54    0.000249           8        33        22 open
 18.05    0.000230          13        18           mprotect
 ...
```

Let's use this to see what gas.py is up to. Let's start with 1 thread:

```
[lowell@te-master gas]$ strace -fc ./gas.py -t1 -p2048 -s 4 -o sym
strace: Process 12878 attached
...
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 59.99    0.083590        1007        83        39 wait4
  7.27    0.010132           5      1872       737 stat
  5.94    0.008277           9       921       471 open
  5.73    0.007990           4      2243           rt_sigprocmask
...
```

Now, for 4 threads:

```
[lowell@te-master gas]$ strace -fc ./gas.py -t4 -p2048 -s 4 -o sym
strace: Process 13006 attached
...
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 99.98  514.903024          40  12806837   3159288 futex
  0.01    0.068111         821        83        39 wait4
```

That's right… it spent 99.98% of execution time in a futex syscall. A futex is a kind of lock that can be requested at the kernel level, in some ways like a mutex.

What's this all about? We are likely the victim of the Python "Global Interpreter Lock" (GIL). See, for instance: Thread State and the Global Interpreter Lock. This is a Python-specific problem, and our code would likely have scaled fairly well in some other languages.

Parallel Programming with MPI

Overview

In the last guide, we learned some of the basics of parallel programming. This guide will focus on parallel programming with the Message Passing Interface (MPI).

Step 1: Setting up the environment

The first thing we need to do is setup Python to be able to run MPI. We do this with a module called mpi4py. To build this module, we will need to be on our newer python version and to have the openmpi3 lmod module loaded.

```
[lowell@te-master mpi-examples]$ module list

Currently Loaded Modules:
  1) autotools   2) prun/1.3   3) gnu8/8.3.0   4) openmpi3/3.1.3   5) ohpc
[lowell@te-master mpi-examples]$ python --version
Python 3.7.3
[lowell@te-master mpi-examples]$ pip install mpi4py
Collecting mpi4py
Installing collected packages: mpi4py
Successfully installed mpi4py-3.0.2
```

Make the following test script just to make sure mpi4py is working:

```
#!/usr/bin/env python3
```

```
from mpi4py import MPI

comm = MPI.COMM_WORLD   # this is the default channel for communicating
 ↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

print("I am {} of {}".format(rank, comm_size))
```

In MPI, the rank is a special sequential numeric ID each MPI task is given. The comm_size is the total number of tasks in our job.

Now let's try to run it with a couple of different settings:

```
[lowell@te-master mpi-examples]$ mpirun -np 1 mpitest.py
...
I am 0 of 1
```

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpitest.py
...
I am 0 of 4
I am 2 of 4
I am 3 of 4
I am 1 of 4
```

Looks like it's working.

Step 2: Basic MPI communications

MPI offers a number of different communications that we can do with other ranks in our system. Those ranks may live on the same node with us, or that may communicate over the network fabric (e.g. Infiniband) that MPI is using.

The communications backend protocol is called the "base transport layer", or "BTL". You'll see references to this in various debugging and warning messages.

MPI communications methods can be broken up into two categories: point-to-point communication and collective communication. First, we will go over a couple of examples that use point-to-point communication. We'll look at collective communication in the next step.

send/recv

Let's start with something very simple. Let's have 2 MPI tasks, and have the first task send the second task a message, then print it.

```
#!/usr/bin/env python3
# mpi-send-recv.py
```

```
from mpi4py import MPI

comm = MPI.COMM_WORLD   # this is the default channel for communicating
 ↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  msg="Hello hello... is there anybody in there?"
  comm.send(msg, dest=1, tag=42)
  print("I (0) sent this msg: " + msg)
elif rank == 1:
  rmsg = comm.recv(source=0, tag=42)
  print("I (1) got this message: " + rmsg)
else:
  print("I'm rank {}.  I don't get to play right now.".format(rank))
```

So, if we're rank 0 we send a message if we're rank 1 we receive it. Both of us print what we sent/received. The tag=# is not required. In more complicated situations, tags can help us be sure we're receiving the message we think we're receiving. Notice that dest= and source= take a rank number as values.

We also were able to pass a Python data object directly through this message. mpi4py works with pickle to serialize objects for transfer. If you do happen to be working with raw byte buffers (e.g. Numpy arrays), there are other routines for that in mpi4py which are probably a little more efficient.

Let's actually run this example:

```
[lowell@te-master mpi-examples]$ mpirun -np 2 mpi-send-recv.py
...
I (0) sent this msg: Hello hello... is there anybody in there?
I (1) got this message: Hello hello... is there anybody in there?
```

Looks like it worked just fine. If we run with more than 2 tasks we get:

```
[lowell@te-master mpi-examples]$ mpirun -np 3 mpi-send-recv.py
...
I (0) sent this msg: Hello hello... is there anybody in there?
I'm rank 2.  I don't get to play right now.
I (1) got this message: Hello hello... is there anybody in there?
```

It's worth noting that send/recv are blocking–they'll wait until they succeed. They have non-blocking equivalents, isend/irecv that will return right away even if they don't send/recieve any data.

Gather

The Gather operation collects all of a certain piece of data to a single task. Let's look at an example:

```
#!/usr/bin/env python3
# mpi-gather.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
 ↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

my_random = randint(0,100)

print("{}: my random int is: {}".format(rank,my_random))

rands = comm.gather(my_random, root=0)
if rank == 0:
  print("0: all of the randoms: {}".format(rands))
```

If we run this:

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpi-gather.py
...
0: my random int is: 42
0: all of the randoms: [42, 48, 83, 57]
1: my random int is: 48
2: my random int is: 83
3: my random int is: 57
```

There are a few things to note about this:

1. *all* of the ranks call comm.gather. The ones that are not root=<#> send data, while the one that is specified by root=<#> collects it. This is common to how all of the collective MPI communications work.
2. when rank=0 gathers the data, that data always includes its own entry.

Allgather

Allgather is similar to gather, except that all of the nodes get all of the information, i.e. N-to-N collections. Let's modify our example with this:

```
#!/usr/bin/env python3
# mpi-allgather.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

my_random = randint(0,100)

print("{}: my random int is: {}".format(rank,my_random))

rands = comm.allgather(my_random)
print("{}: all of the randoms: {}".format(rank, rands))
```

Running this with 4 tasks:

```
[lowell@te-master mpi-examples]$ mpirun -np 4 mpi-allgather.py
...
0: my random int is: 96
1: my random int is: 21
2: my random int is: 98
0: all of the randoms: [96, 21, 98, 80]
1: all of the randoms: [96, 21, 98, 80]
3: my random int is: 80
3: all of the randoms: [96, 21, 98, 80]
2: all of the randoms: [96, 21, 98, 80]
```

MPI allgather is appealing to use if you're used to shared memory systems, because we always have all of the memory as long as we allgather frequently. However, this is often very inefficient, and allgather should probably generally be avoided.

Up until now we've just run these tests on one node. Let's actually make one of them talk over the Infiniband. This will do an allgather with every node generating one random number:

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix
↪  mpi-allgather.py
salloc: Granted job allocation 142
...
2: my random int is: 11
2: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
8: my random int is: 82
8: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
6: my random int is: 99
```

```
6: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
5: my random int is: 98
5: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
7: my random int is: 82
7: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
9: my random int is: 12
9: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
3: my random int is: 85
3: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
1: my random int is: 37
1: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
4: my random int is: 13
4: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
0: my random int is: 55
0: all of the randoms: [55, 37, 11, 85, 13, 98, 99, 82, 82, 12]
salloc: Relinquishing job allocation 142
```

Scatter

A scatter is essentially the inverse of a gather. In a scatter operation, one node has a list of objects, and randomly distributes those objects to all of the nodes. This can be very useful if you want a "work queue" style model. A controlling task can scatter work that needs to be done to worker tasks.

```python
#!/usr/bin/env python3
# mpi-scatter.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
 ↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  d = list()
  for i in range(0,comm_size):
    d.append(randint(1,100))
  print("0: generated {}".format(d))
else:
  d = None

d = comm.scatter(d, root=0)
```

```
print("{}: got {}".format(rank,d))
```

Running:

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix
 ↪  mpi-scatter.py
salloc: Granted job allocation 143
...
1: got 18
0: generated [87, 18, 47, 39, 79, 57, 29, 65, 36, 52]
0: got 87
8: got 36
7: got 65
5: got 57
6: got 29
4: got 79
9: got 52
3: got 39
2: got 47
salloc: Relinquishing job allocation 143
```

Like the other operations, we see that rank=0 also gets a number from the scatter list.

Broadcast

We'll look at one last operation: broadcast. A broadcast operation sends a piece of data to everyone. Let's modify our scatter to be a broadcast:

```
#!/usr/bin/env python3
# mpi-bcast.py

from mpi4py import MPI
from random import randint

comm = MPI.COMM_WORLD   # this is the default channel for communicating
 ↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool

if rank == 0:
  d = list()
  for i in range(0,comm_size):
    d.append(randint(1,100))
  print("0: generated {}".format(d))
else:
  d = None

d = comm.bcast(d, root=0)

print("{}: got {}".format(rank,d))
```

```
[lowell@te-master mpi-examples]$ salloc -N10 -n10 srun --mpi=pmix
↪  mpi-bcast.py
salloc: Granted job allocation 145
...
0: generated [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
0: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
8: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
4: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
1: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
9: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
5: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
6: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
2: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
7: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
3: got [4, 62, 77, 1, 3, 85, 21, 44, 97, 29]
salloc: Relinquishing job allocation 145
```

Those are most of the MPI operations one would generally use, though the spec of operations is quite large. As we can see, MPI allows us to trivially pass data around. MPI also has methods we did not go over for things like task synchronization (e.g. MPI.Barrier, where all tasks pause until everyone has called Barrier). Not only does MPI make it easy to pass data around, it does it efficiently over various different transport layers like Infiniband.

Step 3: The MPI of `gas.py`

Perhaps the easiest way to see how this message passing can be used to make a large application spread over lots of systems is to simply look at an example. Fortunately, we have one. There is an MPI + Threading version of gas.py in the mpi branch:

```
comm = MPI.COMM_WORLD   # this is the default channel for communicating
↪  with our MPI run
comm_size = comm.Get_size() # get the total number of MPI processes
rank = comm.Get_rank()  # this tells us which number we are in the pool


class Sym(threading.Thread):
    def boundary_resolve(self):
        c = 0
        for p in iter(self.w.oob):
            p.boundary_resolve_hard(self.w.box)
            c += 1
        print("\treflected {} particles of {} in the oob list".format(c,
         ↪  len(self.w.oob)), file=sys.stderr)

    def clear_oob(self):
        self.w.oob.clear()
        print("oob list cleared")

    def run_cell(self, c):
        print("\t{:0.4f} comb particles into cells".format(since(self.w)),
         ↪  file=sys.stderr)
        c.comb_particles()
        print("waiting for comb to complete")
        self.b.wait() # once we're done combing, we have to clear all
         ↪  leftover oob particles; they must have gone to a different
         ↪  rank
        print("\t{:0.4f} start integrate".format(since(self.w)),
         ↪  file=sys.stderr)
        c.integrate()
        print("\t{:0.4f} start collide".format(since(self.w)),
         ↪  file=sys.stderr)
        c.collide()
        print("\t{:0.4f} start boundary check".format(since(self.w)),
         ↪  file=sys.stderr)
        c.boundary_check()

    def print(self):
        for c in self.cells:
            c.print(self.w.out)

    def run(self):
        for i in range(0,threads):
            zrange = (self.w.box.z.high - self.w.box.z.low)/threads
            yrange = (self.w.box.y.high - self.w.box.y.low)/comm_size # we
             ↪  want things comm_size * threads
            # We slice the z-dimension by number of threads, and the y
             ↪  dimension by comm size
            c = Cell(i, self.w, int(parts/threads), Box(self.w.box.x,
             ↪  Range(rank*yrange, (rank+1)*yrange), Range(i*zrange,
             ↪  (i+1)*zrange)))
            self.cells.append(c)

        # create a cell
```

```
print("Running with {} threads, {} comm_size".format(threads,
 ↪  comm_size))
print("My MPI rank is {}".format(rank))
print("{:0.4f} running with {} particles for {} steps, output to
 ↪  {}".format(since(self.w), parts, steps, out.name),
 ↪  file=sys.stderr)

print("--- ts=0", file=out)
self.print()

for i in range(1, steps+1):
    tds = list()
    for c in self.cells:
        print("{:0.4f} timestep={},
         ↪  particles={}".format(since(self.w), i, len(c.parts)),
         ↪  file=sys.stderr)
        tds.append(threading.Thread(target=self.run_cell,
         ↪  args=(c,)))

    self.b.reset()
    for td in tds:
        print("starting thread")
        td.start()

    for td in tds:
        td.join()

    all_oob = comm.gather(self.w.oob, root=0)
    print(all_oob)
    self.w.oob: List[Particle] = []
    for i in all_oob or []:
        for j in iter(i) or []:
            self.w.oob.append(j)

    if rank == 0:   # rank 0 gathers all oob and computes boundary
     ↪  conditions for the whole world
        print("\t{:0.4f} resolve out-of-bounds
         ↪  particles".format(since(self.w)), file=sys.stderr)
        self.boundary_resolve()

    # how broadcast the list to everyone else, post boundary
     ↪  conditions
    self.w.oob = comm.bcast(self.w.oob, root=0)
```

```
            print("\t{:0.4f} start print".format(since(self.w)),
             ↪  file=sys.stderr)
            print("--- ts={}".format(i), file=out)
            self.print()

        print("{:0.4f} done".format(since(self.w)), file=sys.stderr)
        out.close()


    def __init__(self):
        self.b = threading.Barrier(threads, action=self.clear_oob)
        self.cells = list()
        self.w = World(steps=steps, out=out, parts=parts, cells=cells)
```

Our simple gas.py simulator clearly got a little more complicated, but not dramatically so. There are a couple of things we had to do to map in MPI:

1. we divide cells now not only in the z-axis for threads, but also in the y-axis for rank (in exactly the same way). So, our cells are like long rectangular rods.
2. when a particle gets on the out-of-bounds list when the cell does a boundary check, we need to potentially transfer it to a neighbor. Here's how we do that:
    1. we gather all oob lists to rank=0
    2. rank=0 does total world size boundary conditions
    3. once boundary conditions are satisfied, rank=0 broadcasts the (potentially modified) oob list to all ranks
    4. each cell combs for its particles as usual
    5. . . . but there's a complication

When the cells comb for their particles, because some particles will belong to other ranks, any particular rank will not have an empty list of oob particles by the end of the combing process. For this reason, we have to clear the list manually. But, to do this, we need to be sure that all cells (threads) in our rank are done combing.

To ensure this, we implement a thread "barrier". The barrier, once called, blocks until all threads have called it. Once all threads have called it, it executes the action= specified by:

`self.b = threading.Barrier(threads, action=self.clear_oob)`

This is just a function that clears the oob list.

So, we see that going to MPI + Threads required a bit more bookkeeping, but ultimately isn't that much harder than our earlier versions.

Here's how the lines of code break down for code in Sym:

- single-threaded: 29 lines
- multi-threaded: 61 lines
- MPI + threaded: 84 lines

Finally, let's run the gas.py in MPI mode.

```
[lowell@te-master gas]$ time mpirun -np 2 ./gas.py -t2 -p 1024 -s4 -o sym
...
real    0m12.349s
user    0m19.839s
sys    0m0.416s
```

Try running it across more than one node. Does it work best to use MPI or threads or both?

Some experimenting shows it doesn't scale very well to large ranks. Trying to run across the whole cluster, for instance, tends to break.