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