Configuring a Highly Available service on FreeBSD

– Part 2: CARP and devd

In the first part of this series, we learned how to make high availability (HA) storage on FreeBSD using HAST. We learned how to control HAST and how to recover from failures. However, all those actions were still manual actions. In this second part of the series, we will learn how two basic building blocks, CARP and devd, work and how we can use them in the final part of our series to automate the failover of our NFS server.

What you will learn…

- How to configure CARP on FreeBSD
- How to use devd to take action on kernel events

What you should know…

- How to login to FreeBSD
- How to edit files on FreeBSD
- Basic understanding of network configuration
- The nfs-01 and nfs-02 machines from part 1

CARP stands for common address redundancy protocol and makes it possible to share an IP (IPv4 and/or IPv6) address between multiple hosts in so called 'redundancy groups'. The IP that is shared between the hosts in the redundancy group resides on the master host for that group. In case the master goes down, the other members (backups) in the redundancy group will elect a new master. This master will then 'take' the shared IP.

That, of course, sounds nice, but how does that help us? Well, to implement our failover NFS service, we need an IP address for this service to reside on the host that will service the NFS requests. The host that will service the NFS request would be the primary HAST node. Also, in case of a HAST failover, we would like the service IP to switch to the new primary HAST node. So, if we are able to keep the CARP master state and the HAST primary state in sync with each other, we would always have the shared IP, which we can use for the NFS service, on the host that is the primary HAST node.

How to configure CARP

CARP can be configured by using the *ifconfig* command as described in listing 1. Note that in our example, setup nfs-01 will have the IP 192.168.254.1 and nfs-02 will have the IP 192.168.254.2. Both with a /24 netmask.

The first command for nfs-01 in Listing 1 creates a carp interface called *carp0* on that host. The second command configures this newly created *carp0* interface with the correct parameters. The first parameter *vhid* is the virtual host ID, which uniquely identifies the redundancy group on the network and therefore should be the same on all hosts in the same redundancy group. In our example, we use a *vhid* of 1. The second parameter pass is used to authenticate the carp advertisements and is in our case set to *bsdmag*. This parameter should also be the same on all hosts in the same redundancy groups. Although the *pass* parameter is optional, it is wise to set it, otherwise machines not part of the redundancy group can easily send out bogus carp traffic to disrupt our redundancy group. The third parameter is *advbase*, which specifies the base advertisement interval in seconds. These advertisements are needed to determine if the master is still up and if not to elect a new master. The fourth parameter *advskew* is closely related to the *advbase* parameter; when set, it adds a small amount of time to *advbase* so that advertisements are sent out a little less frequently than specified by *advbase*. This fourth parameter differs in our example for nfs-01 and nfs-02. It is higher for nfs-02 so that nfs-01 will become the master if both hosts come online at the same time, because nfs-01 will send out its advertisements more frequently. The last parameter specifies the shared IP to use and the network it resides on. In our case, the shared IP is 192.168.254.100 with a /24 netmask. This IP will become active on the master on the interface that is in the same network as specified for the carp0 interface. If, for example, nfs-01 is the master, the shared IP 192.168.254.100 will become available on the same interface as 192.168.254.1 as that interface is in the same network.

Listing 1. *configuring CARP on our hosts*

```
nfs-01# ifconfig create carp0
nfs-01# ifconfig carp0 vhid 1 pass bsdmag advbase 1
   advskew 10 192.168.254.100/24
nfs-01# ifconfig carp0
  carp0: flags=49<UP,RUNNING> metric 0 mtu 1500
         inet 192.168.254.1 netmask 0xffffff00
         carp: MASTER vhid 1 advbase 1 advskew 20
nfs-02# ifconfig create carp0
nfs-02# ifconfig carp0 vhid 1 pass bsdmag advbase 1
   advskew 20 192.168.254.100/24
nfs-02# ifconfig carp0
  carp0: flags=49<UP,RUNNING> metric 0 mtu 1500
         inet 192.168.254.2 netmask 0xffffff00
         carp: BACKUP vhid 1 advbase 1 advskew 10
```

The third command not only shows us the configuration of our *carp0* interface, but also shows whether the interface is in the master or in the backup state in the line starting with *carp:*. Note that the password used is not visible. The configuration of the *carp0* interface on the nfs-02 is analog to the configuration of the *carp0* interface on the nfs-01 with the earlier mentioned difference of the advskew parameter.

Making CARP reboot proof

Now that we know how to configure CARP we want to make sure that our configuration becomes reboot proof. This can be done by adding a few lines to /etc/rc.conf. In listing 2 you can find the lines we would need to add to /etc/rc.conf on the nfs-01 server. The first line makes sure our *carp0* device will be created on boot. The second line configures the *carp0* interface and is identical to the parameters we passed to the ifconfig command for *carp0* earlier. It is left as an exercise for the reader to find the correct configuration for the nfs-02 server.

Listing 2. *Making the CARP configuration reboot proof on nfs-01*

```
cloned_interfaces="carp0"
ifconfig_carp0="vhid 1 pass bsdmag advbase 1 advskew 10
   192.168.254.100/24"
```

Testing CARP

After we have made our CARP configuration reboot proof, it is good to perform some basic tests to see whether the failover of the shared IP works as expected. First we will force a switch of the shared IP from our current master (nfs-01) to nfs-02. When that is complete and nfs-02 has indeed become the new master we will force the master back to the nfs-01. In addition to testing, the commands in listing 3 that describe these actions are also useful in the case when a manual switch has to be forced. Please be especially aware of the host you have to execute the commands on to trigger the failover. An important note to make is that in case you are building this setup on a virtual platform, broadcast traffic should be allowed for the virtual machines or CARP won't work. Allowing broadcast traffic is not the default setting for all virtualisation solutions.

What is DEVD?

Devd is the device state change daemon and it is a system daemon that runs in the background and hooks into the *devctl* device driver. When a change occurs in the device configuration tree, this device driver will pass this information to devd. Devd will parse this message and will look into its action list for an action to execute. This way devd provides a way to have userland programs run when certain kernel events happen. The default configuration file for devd is `/etc/devd.conf`. By default this file includes the options to also scan the `/etc/devd` and `/usr/local/etc/devd` directories for devd configuration files.

Listing 3. *Testing the CARP failover*

```
Moving the master from nfs-01 to nfs-02 (commands
    executed on nfs-01)
nfs-01# ifconfig carp0 down
nfs-01# ifconfig carp0 up

Checking the status on both hosts (commands executed
    on nfs-01 and nfs-02)
nfs-01# ifconfig carp0
nfs-02# ifconfig carp0

Moving the master from nfs-02 to nfs-01 (commands
    executed on nfs-02)
nfs-02# ifconfig carp0 down
nfs-02# ifconfig carp0 up

And again checking the status on both hosts (commands
    executed on nfs-01 and nfs-02)
nfs-01# ifconfig carp0
nfs-02# ifconfig carp0
```

Devd syntax

To explain the syntax of devd we will make a slight side step by looking at the devd configuration shown in Listing 4. What this configuration does is log a message to syslog when a USB device is attached. Let's inspect this configuration a little bit further. The first line *notify 0* indicates that an action should be taken when the kernel sends an event notification to the user land. The priority of this rule is 0. This priority is used to decide which action to take when more than one rule matches. If more than one rule matches the rule with the lowest number is executed. To restrict the cases in which our action will be executed we use the *match* clauses on line 2 till 4 to restrict it. Line 2 matches the event message against the system it is coming from, in this case the USB system. So all events that are not from the USB system will not trigger the action. The next line restricts the action to a subsystem of the USB system. In this case it is the *interface* subsystem, so the event should come from a USB interface to trigger our action. The last match rule of Listing 4 restricts the type of event, in this case the attachment of a device. Last but not least, we have line 5, which specifies the action to execute. In this case we log a message to syslog to notify us that a USB device has been attached, but an action line can call every command you like. More information about all the systems, subsystems, types and action you can handle with devd can be found in the *devd.conf* manual page.

Listing 4. *A devd configuration for USB events*

```
notify 0 {
      match "system"        "USB" ;
      match "subsystem"  "INTERFACE" ;
      match "type"            "ATTACH" ;
      action "logger USB device attached" ;
} ;
```

Listing 5. *Configuring devd for CARP*

```
notify 30 {
      match "system"        "IFNET" ;
      match "subsystem"  "carp*" ;
      match "type"            "LINK_UP" ;
      action "logger -t bsdmag $subsystem device is
   UP" ;
} ;

notify 30 {
      match "system"        "IFNET" ;
      match "subsystem"  "carp*" ;
      match "type"            "LINK_DOWN" ;
      action "logger -t bsdmag $subsystem device is
   DOWN" ;
} ;
```

Configuring devd for CARP

Now that we have a basic grasp of how to use devd to take actions on kernel events we can start to configure devd to handle events originating from our CARP interfaces. In listing 5 we see a configuration that will log to syslog when we receive a LINK_DOWN or a LINK_UP event from our carp0 interface. Because a CARP device is a network system, the system we have to use in our match rule is IFNET. Noteworthy is the wildcard match we use in the subsystem, hence the action will run for an event on any carp interface that matches the type. To separate between the link going up and the link going down we created 2 statements, one for the LINK_UP and one for the LINK_DOWN event. Also interesting is the action line we use. Again, we use *logger* to log a message to syslog, but we also use the *$subsystem* variable available to log the exact subsystem that the event came from, so in the log we will see which interface generated the event. By putting the configuration from listing 5 in `/etc/devd/hast.conf` and by restarting devd with *service devd restart* we make sure it will be used by devd.

Testing our devd configuration

Testing of our devd configuration is more or less analog to the initial testing of our CARP configuration, so we can use the commands from listing 3 again. However, to see if our devd recipe for CARP worked, we should not only check the status of our *carp0* interface with *ifconfig carp0*, but also check `/var/log/messages` to see if the log messages we configured in listing 5 are indeed written to the syslog correctly, so we are sure devd is configured correctly. Take good note of when a CARP interface sends the LINK_UP type and when it sends the LINK_DOWN type of event. You will see that the CARP interface sends the LINK_UP message via devd only when it becomes the master and the LINK_DOWN message when it goes down and when it becomes the backup.

Conclusion

In this second part of the series we introduced CARP and devd. We learned how to configure CARP and how to make an IP highly available with it. We also learned what devd is and how to take actions on kernel events by using devd. Especially, we learned how to run a script from devd in case of a CARP failover. Now that we know how to configure HAST, CARP and devd we can put all these building blocks together in the final part of our series in which we will create the highly available NFS server and the failover script to call from devd.

Questions received from readers

Q: How highly available is HAST when a filesystem check can take ages on a large filesystem?

During the filesystem check HAST is of course unavailable. But you really should do this check, to be sure your filesystem is in a consistent state. Otherwise you might run into problems later that cause much more downtime than the filesystem check would take. To reduce the time spent filesystem checking it is also good practice to always use a journaling filesystem on your HAST devices. One important point to keep in mind is also that highly available does not mean *always* available, so yes, in case of a node failure you probably will have some downtime, but significantly less than when you would need to rebuild your machine and restore from backup. Also your data-loss will probably be significantly less.

Q: But what if I do need my filesystem to always be available?

In that case you should probably look into gluster (RedHat Linux), LustreFS or CEPH, which are clustered/distributed filesystems but all have a focus on Linux unfortunately.

JEROEN VAN NIEUWENHUIZEN

Jeroen van Nieuwenhuizen works as a unix consultant for Snow. His free time activities beside playing with FreeBSD include cycling, chess and ice skating.