Adding a DNS Proxy to 128T

Many edge routers at small office/remote branch locations offer a DNS caching proxy feature, where the router offers address to site clients using DHCP with its own address as the DNS server. This document demonstrates how to incorporate the popular, ad-blocking DNS server *pi-hole* into your 128T to offer these same benefits, as well as improving the client experience.

> Note: for a comprehensive list of the features and benefits of Pi-hole, visit www.pi-hole.net

Overview

We will use *Docker* to run Pi-hole in a container on the same host platform as the 128T. This affords us the ability to upgrade and/or remove it simply with minimal impact to the 128T configuration or host platform. As with other applications running on the Linux host platform, we will communicate between Pi-hole and 128T/the outside world using *Kernel Network Interfaces* (KNI) to shuttle packets back and forth.

Requirements

This overview assumes you are running your 128T on a hardware platform that meets 128 Technology's minimum requirements. The additional overhead of running a DNS proxy is nominal, and should not impact normal operation of your 128T.

Software Dependencies

- Docker is used to retrieve and manage the lifecycle of our DNS server. Install docker from the Linux command line:

  ```
  sudo dnf install docker -y
  ```

Configuration Dependencies

- Pi-hole relies on using its own webserver *lighttpd* for management purposes. This will collide with the 128T's internal web server. Given that it is fairly uncommon to manage a router individually using its GUI (as opposed to via its Conductor), you can disable the webserver on an indivudal 128T within its configuration:

```
admin@hari.burlington# config auth router burlington system services webserver
enabled false
admin@hari.burlington# commit
Are you sure you want to commit the candidate config? [y/N]: y
✔ Validating, then committing...
Configuration committed
```

Launching Containers

Due to its packet forwarding architecture, 128T pins one or more CPU cores for exclusive use when it starts. Likewise, Docker will assess the platform's CPU resources when it starts; by default, it inhibits 128T from dedicating core(s) to itself. In this section we'll limit the number of cores that Docker will see, to ensure that there will be dedicated CPU cores available for the 128T's packet forwarding engine.

Creating a Docker cgroup service

To limit the number of CPU cores available to Docker, we'll create a `docker-cgroup.service` file, which will constrict the `cpuset.cpus` value to a small number of cores.

To see how many cores you have available on your machine, use `lscpu` as seen below:

```
[root@labsystem2 ~]# lscpu | grep list
On-line CPU(s) list:   0-3
```

This is a four core machine. We'll limit Docker to one core, leaving the rest available to 128T and the Linux operating system.

> Note: one core is ample room for Pi-hole to run. If you intend on running more containers, you may need to adjust this value. That is beyond the scope of this how-to guide, however.

Create a file at `/etc/systemd/system/docker-cgroup.service` with the following contents:

```
[Unit]
Description=Create cgroup for docker

[Service]
Type=oneshot
ExecStart=/bin/sh -c ' \
        mkdir -p /sys/fs/cgroup/cpuset/dockercg.slice; \
        /bin/echo 3 > /sys/fs/cgroup/cpuset/dockercg.slice/cpuset.cpus; '

ExecStop=/bin/sh -c 'rmdir /sys/fs/cgroup/cpuset/dockercg.slice'
RemainAfterExit=true
```

```
[Install]
RequiredBy=docker.service
```

The important pieces to point out are the value after `/bin/echo`; in the sample above, we're using `3`, to reserve CPU core 3 to Docker. Adjust this as necessary for your environment. The other piece is the last line, which causes Linux's `systemd` to load this service before it loads `docker.service`. This ensures that the Docker engine is limited in the number of CPUs before it starts.

Configuring Docker to use the cgroup

We need to change the standard Docker configuration to reference this cgroup we've created. The Docker configuration is located at `/etc/docker/daemon.json`. This file (which is `{ }` by default), needs to contain the following:

```
[root@labsystem2 ~]# cat /etc/docker/daemon.json
{ "cgroup-parent" : "dockercg.slice" }
```

Enable the Services

To enable Docker (so that it starts at boot), use the following commands:

```
[root@labsystem2 ~]# systemctl daemon-reload
[root@labsystem2 ~]# systemctl enable docker-cgroup
[root@labsystem2 ~]# systemctl enable docker
```

These changes require a reboot to take effect.

Verifying Docker is Running

After rebooting, use `systemctl status docker` to ensure it is running:

```
[root@labsystem2 ~]# systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; vendor
preset: disabled)
   Active: active (running) since Sun 2019-04-28 13:41:06 EDT; 5h 6min ago
     Docs: http://docs.docker.com
 Main PID: 1278 (dockerd-current)
   CGroup: /system.slice/docker.service
           ├─ 1278 /usr/bin/dockerd-current --add-runtime docker-runc=...
           ├─ 1365 /usr/bin/docker-containerd-current -l ...
```

Here you can see it is enabled and active.

> Note, you may also want to check `systemctl status 128T` to make sure that 128T is also enabled and running.

The DNS Container

The DNS container we're using is maintained by the people that produce Pi-hole. We'll tweak the sample `docker_run.sh` from their Github repo to suit our needs here.

```
[root@hari ~]# mkdir -p /srv/docker/pihole
[root@hari ~]# cd /srv/docker/pihole
[root@hari pihole]# wget https://github.com/pi-hole/docker-pi-
hole/blob/master/docker_run.sh
```

Aside from modifying the timezone (line 10), this file is otherwise ready to go as-is.

Start Pi-hole by running this script.

```
[root@hari ~]# cd /srv/docker/pihole
[root@hari pihole]# ./docker_run.sh
```

You can verify it is working by using the various `docker` commands:

```
[root@hari ~]# docker ps
CONTAINER ID        IMAGE                  COMMAND             CREATED
  STATUS                 PORTS
                                  NAMES
3341855b50a3        pihole/pihole:latest   "/s6-init"          2 days ago
Up 2 days (healthy)   0.0.0.0:53->53/tcp, 0.0.0.0:80->80/tcp, 0.0.0.0:443-
>443/tcp, 0.0.0.0:53->53/udp, 67/udp   pihole
```

Configuring DNS

There are several aspects to configuring DNS once the proxy is running locally:

- Configuring the 128T system to use its own DNS
- Configuring the DNS proxy to reference upstream DNS servers
- Configuring the 128T to allow external client use

We will cover each of these in turn.

Configuring the 128T to use its own DNS

There are many ways to configure a system to reference DNS, but ultimately it boils down to a list of `nameserver` entries in the file `/etc/resolv.conf`. On our reference system, the interfaces are managed by the Linux application NetworkManager. Thus, we'll use NetworkManager (for example, `nmtui`) to set the DNS server to use the localhost: 127.0.0.1.

```
[root@labsystem2 pihole]# cat /etc/resolv.conf
# Generated by NetworkManager
nameserver 127.0.0.1
```

> Note: the contents of this file may be impacted by 128T when it starts. Changes to NetworkManager configuration or `/etc/resolv.conf` should be done while 128T is stopped on the host.

Configuring the Proxy for Upstream DNS

To configure the proxy's DNS, there are two portions: configure the Pi-hole with its upstream DNS servers, and establish a route from Pi-hole to the chosen servers.

There are multiple ways to configure Pi-hole's upstream DNS servers. You can edit the `docker_run.sh` script that launches the container (or if you choose to use docker-compose, its YAML file), or you wait until Pi-hole is up and running and configure the upstream DNS servers using its web interface.

The Pi-hole referenced in this document will mount `./etc-pihole/` to `/etc/pihole/` on the container, and this is where the upstream DNS servers are stored.

Adding Routes to the Upstream Servers

First, we need to get the packets from Pi-hole into 128T. This is done using the same `kni254` interface that we use for `host-service` access (see below). Adding routes from Pi-hole up to 128T is covered in the document "Linux Host Networking Through 128T." (The basic premise is to add a `route-kni254` file within `/etc/sysconfig/network-scripts` that sets the KNI device as the default route for Linux.)

> Note: some configurations, particularly older configurations, use a handcrafted KNI interface in addition to `kni254`. To use a hand-configured KNI interface, add a corresponding route from Linux to use the KNI.

Once the Linux route is in place, you'll need to have a *service* to get to the DNS server(s) you selected, and that service needs an access policy to allow the `_internal_` tenant — the one automatically assigned to all packets arriving on the `kni254` interface.

While many configurations use a "quad 0" service (0.0.0.0/0) to represent a system-wide default route, it is advisable to have a specific service for access to the upstream DNS servers. By way of example:

```
*admin@labsystem1.fiedler (service[name=cloudflare-dns])# show
name                    cloudflare-dns
scope                   private

transport               udp
    protocol    udp

    port-range  53
        start-port  53
    exit

exit
address                 1.1.1.1/32
address                 1.0.0.1/32

access-policy           _internal_
    source      _internal_
    permission  allow
exit
share-service-routes  false
```

Add a `service-route` for this traffic on the system we're installing Pi-hole on, and this should be enough.

Configuring External DNS Access

To allow external access to the DNS proxy, we'll use a *host-service* on the network-interface(s) we wish to expose DNS service to. A `host-service` is a built-in means by which a 128T can take packets on an "external" (forwarding) interface and funnel them down to the Linux host operating system (the "host" in `host-service`) via a KNI that it creates at system launch.

Configuring external DNS access is therefore as simple as adding a host-service on the interface for 53/UDP (and possibly 53/TCP). To expose the web administration interface for Pi-hole, you'll need to add one for 80/TCP.

> Note: the built-in `host-service` for `web` only opens port 443, not port 80.

After adding this configuration, clients should be able to connect to the interface's IP at 53/UDP and these packets will find their way to Pi-hole.

Testing DNS

In this particular example, we have Pi-hole running on a 128T host platform, and that platform's LAN interface is 192.168.1.1.

From an external device:

```
k9:~ ptimmons$ dig @192.168.1.1 www.128technology.com

; <<>> DiG 9.10.6 <<>> @192.168.1.1 www.128technology.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 40101
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1452
;; QUESTION SECTION:
;www.128technology.com.   IN  A

;; ANSWER SECTION:
www.128technology.com.  139 IN  A 35.185.239.152

;; Query time: 88 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Tue Apr 30 21:33:05 EDT 2019
;; MSG SIZE  rcvd: 66
```

From the 128T's host platform:

```
[root@labsystem2 ~]# dig @127.0.0.1 community.128technology.com
;; Truncated, retrying in TCP mode.

; <<>> DiG 9.9.4-RedHat-9.9.4-61.el7_5.1 <<>> @127.0.0.1
community.128technology.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 44653
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1452
;; QUESTION SECTION:
;community.128technology.com. IN  A

;; ANSWER SECTION:
community.128technology.com. 86 IN  CNAME 128technology.connectedcommunity.org.
128technology.connectedcommunity.org. 134 IN A  52.70.138.19
128technology.connectedcommunity.org. 134 IN A  52.6.165.57
```

```
;; Query time: 94 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Tue Apr 30 21:33:57 EDT 2019
;; MSG SIZE  rcvd: 138
```