@padukajorat

Fiddling with FreeBSD - **Netgraph**

What is netgraph?

Netgraph(4) in FreeBSD is a powerful and flexible in-kernel networking subsystem.
It offers a method for combining protocol and link-level drivers, a modular approach to implementing new protocols, and a common framework for kernel entities to communicate.

The fundamental concept in Netgraph(4) is the use of nodes and hooks, where each node has a specific type, and nodes are interconnected through hooks. each node type is documented in its man pages, `$ apropos netgraph` OR `$ man -k netgraph`.

In this article, we will explore `ng_ether(4)`, `ng_bridge(4)`, and `ng_eiface(4)`, along with some hands-on examples.

What do you need to follow along?

Freebsd installed on laptop, desktop or in a vm and an extra network interface. if using primary interface on VM you will loose connectivity untill upper hook of ng_ether is connected, best use second interface.

Let's Start...

To interact with Netgraph, the `ngctl(8)` utility is provided, which allows you to create nodes and connect hooks.
There are two ways to use `ngctl(8)` interactively and non-interactively.

Before using `ngctl(8)`, let's first list the kernel modules currently loaded on the system.

`$ kldstat`

If you haven't run `ngctl(8)` previously, you may not see any kernel modules starting with `ng_*`. However, if you're using a desktop environment, you might see modules like `ng_ubt` and `ng_bluetooth` already loaded.

Let's run `ngctl(8)` now and observe its output, as well as the output of `kldstat`

`$ doas ngctl list` **OR** `# ngctl lists`

```
~ » doas ngctl list
There are 1 total nodes:
  Name: ngctl1021       Type: socket            ID: 00000002   Num hooks: 0
```

`$ kldstat`

Below modules related to Netgraph have been dynamically loaded.

```
6    1 0xffffffff83021000     38f8 ng_socket.ko
7    1 0xffffffff83025000     abb8 netgraph.ko
```

The `ng_socket(4)` node type enables user-mode processes to interact with the kernel's Netgraph(4) networking subsystem through the BSD socket interface. in this case `ngctl(8)` utility has created socket for us.

observe that `$ doas ngctl list` output where Name and ID changes every time you run this command.

Now let us load the `ng_ether(4)` kernel module and observe the `ngctl list` command output.

`$ doas kldload ng_ether.ko`

`$ kldstat`

```
8    1 0xffffffff83030000     31e0 ng_ether.ko
```

`$ doas ngctl list` **OR** `$ doas ngctl ls`

```
~ » doas ngctl list
There are 3 total nodes: (1)                                   (2)
  Name: vtnet0          Type: ether             ID: 00000006   Num hooks: 0
  Name: vtnet1          Type: ether             ID: 00000007   Num hooks: 0
  Name: ngctl1241       Type: socket            ID: 00000008   Num hooks: 0
```

(1)

Each node has a type, which is a static property of the node determined at node creation time.

(2)

Nodes are connected by pairs of hooks, allowing bidirectional data flow. Each node can have multiple hooks and assign its own meaning to them.

Once the `ng_ether` module is loaded into the kernel, a node is automatically created for each Ethernet interface on the system. Each node will try to name itself after the associated interface.

As of now we are running commands non-interactively, let us do some interactive communication with `ngctl`.

`$ doas ngctl` **OR** `# ngctl`

Your prompt should now display the output of the available commands in interactive mode.

Let's view few of them.

```
+ types
There are 2 total types:
      Type name   Number of living nodes
      ---------   ----------------------
          ether       2
         socket       1
+
```

```
+ show vtnet1:
  Name: vtnet1          Type: ether             ID: 00000007   Num hooks: 0
+
```

OR

```
+ info vtnet1:
  Name: vtnet1          Type: ether             ID: 00000007   Num hooks: 0
+
```

OR

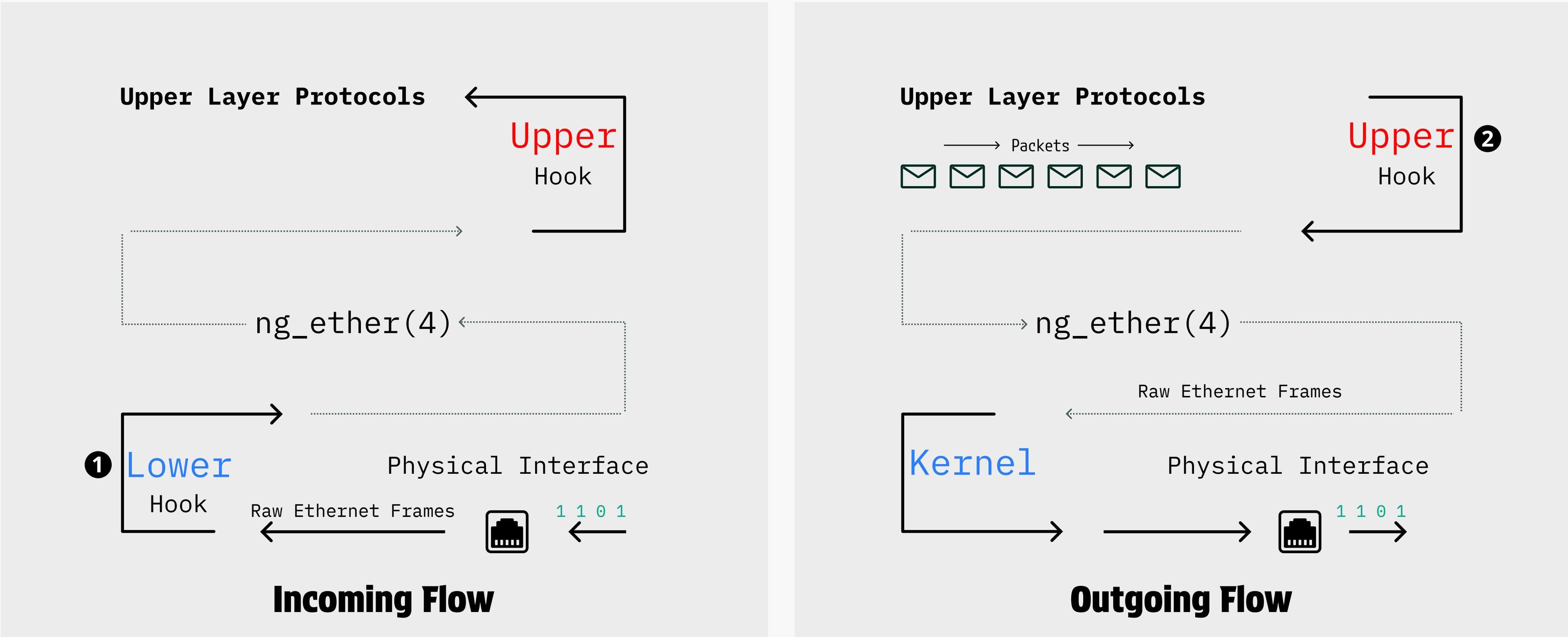
```
+
+ inquire [007]:
  Name: vtnet1          Type: ether             ID: 00000007   Num hooks: 0
```

```
+ quit
```

Let's learn `ng_ether(4)`, `ng_bridge(4)` and `ng_eiface(4)`.

Let's make productive use of `ng_ether(4)` by creating `ng_bridge(4)` and linking it to **ng_ether's lower hook**. First, let's take some time to understand what `ng_ether` and `ng_bridge` are.

`ng_ether(4)` provides three hooks: **lower**, **upper**, and **orphans**.



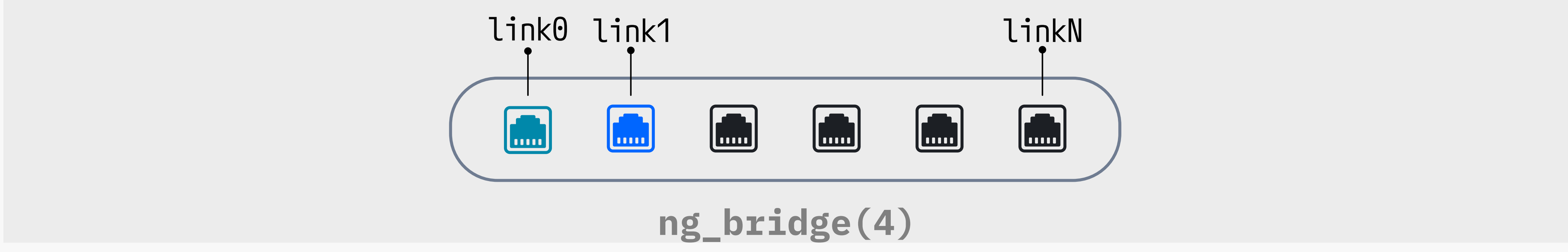
Lower Hook - **Incoming Flow** (1)

The lower hook connects to the raw Ethernet device. When connected, all incoming packets are forwarded to this hook, bypassing the kernel for upper-layer processing.

Upper Hook - **Outgoing Flow** (2)

The upper hook connects to the upper protocol layers. Outgoing packets are forwarded here instead of being sent by the device, and writing to it causes a raw Ethernet frame to be received by the kernel.

`ng_bridge(4)` provides unlimited hooks: `link0, link1...linkN`



Let's do hands on `ng_ether(4)` and `ng_bridge(4)`

`$ doas ngctl mkpeer vtnet1: bridge lower link0`

`$ doas ngctl list`

`$ doas ngctl show vtnet1:`

```
~ » doas ngctl show vtnet1:
  Name: vtnet1          Type: ether             ID: 00000007   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  lower           <unnamed>       bridge       0000000d        link0
```

The output should display the `local hook`, namely the `lower hook`, which is connected to the peer type `bridge` on the `peer hook link0`.
However, if you check the peer name field, it shows "<unnamed>". Let's assign a name to our bridge.
Two ways we can do this either **interactive mode** or **non-interactive mode**.

Interactive mode

`$ doas ngctl` OR `# ngctl`

`+ ls`

```
+ ls
There are 4 total nodes:
  Name: vtnet0          Type: ether             ID: 00000003   Num hooks: 0
  Name: vtnet1          Type: ether             ID: 00000004   Num hooks: 1
  Name: <unnamed>       Type: bridge            ID: 00000007   Num hooks: 1
  Name: ngctl1112       Type: socket            ID: 0000000a   Num hooks: 0
+
```

`+ show vtnet1:` OR `+ show [007]:`

```
+ show vtnet1:
  Name: vtnet1          Type: ether             ID: 00000004   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  lower           <unnamed>       bridge       00000007        link0
+
```

`+ name [007]: firstbridge` OR `+ name vtnet1:lower bridge0`

`+ show vtnet1:`

`+ show bridge0:` OR `+ show firstbridge:`

Non-interactive mode

`$ doas ngctl list`

Assuming you have ID: `000000b` of the bridge

`$ doas ngctl name "[00b]:" firstbridge`

OR

`$ doas ngctl name vtnet1:lower bridge0`

Now we are going to connect our `ng_ether(4)` **upper** hook to `ng_bridge(4)` **link1**

`$ doas ngctl connect vtnet1: bridge0: upper link1`

`$ doas ngctl show bridge0:` OR `$ doas ngctl show vtnet1:`

`$ doas ngctl ls -ln`

With that, the bridge setup with vtnet1 is complete. Now, let's create another node type called `ng_eiface(4)` supports single hook called `ether`, which is a generic Ethernet interface, and named as `ngeth0`, `ngeth1`, and so on.

`$ doas ngctl mkpeer . eiface ether ether`

here "." is local node and you can exclude it. you should see `ngeth0` of type `eiface` created.

`$ doas ngctl list`

let's connect our `ngeth0 ether` hook to `bridge0 link2`

`$ doas ngctl connect ngeth0: bridge0: ether link2`

`$ doas ngctl msg vtnet1: setpromisc 1`   ⟵ Read `ng_ether(4)` man page

With that our `ngeth0` is connected to `bridge0` on `link2`, let's create jail and assign `ngeth0` to our first **jail**.

`$ doas jail -c name=first host.hostname=first.home.arpa vnet persist`

`$ doas ifconfig ngeth0 vnet first`

lets assign ip address to our first jail. assign this based on your network not what i mention below.

`$ doas ifconfig -j first ngeth0 172.16.80.151/24`

Try pinging from host, you should now have working jail setup. you can create ngeth1 and connect it to link3 of bridge0 and assign ngeth1 to second jail.