Introduction to NETGRAPH on FreeBSD Systems

Extended Abstract
Tutorial Slides
'All About Netgraph'
man 8 ngctl
man 8 nghook
man 4 netgraph

March 22 – 23, 2012 at AsiaBSDCon
Tokyo University of Science, Japan

Dr. Adrian Steinmann <ast@marabu.ch>

Introduction to NETGRAPH on FreeBSD Systems

Summary

FreeBSD's NETGRAPH infrastructure can be understood as customizable "network plumbing". Its flexibility and the fact that this infrastructure runs in the kernel makes it an attractive enabling technology where time-to-market, agility, and performance are important.

The goal of the tutorial is to become familiar with FreeBSD's NETGRAPH framework and the available NETGRAPH kernel modules. The participants will gain insight and understanding for which projects lend themselves well to NETGRAPH solutions. A number of examples are shown which can be used as a starting point for new NETGRAPH projects.

In the first part of the tutorial, the NETGRAPH nodes, hooks, and control messages are described and the command syntax is explained via demonstrations on simple examples. Participants learn how they can describe a network connection in terms of its underlying protocols and how to express a solution using NETGRAPH terminology.

The second part of the tutorial investigates frequently used NETGRAPH nodes and shows how they interconnect to create network protocols. More complex NETGRAPH examples including VLAN bridges, UDP tunnels, and the Multi-link Point-to-Point daemon are described. Guidelines and resources for developing custom NETGRAPH modules are surveyed.

Tutorial Outline

Since its introduction to FreeBSD 3.x over a decade ago, the NETGRAPH system is being continuously maintained and developed by the community as can be seen by the constant addition of new `ng_*` kernel modules supporting additional networking features. The tutorial is structured along the following outline:

- History and motivation
  - From TTY device driver to System V STREAMS
  - Transport Protocol Multiplexors and Graphs
  - NETGRAPH Platforms, Software Licensing
  - Evolution on FreeBSD
  - Important Reading Resources

- How to build a NETGRAPH
  - Prerequisites
  - Creating Nodes
  - Connecting Nodes and Creating Edges

- Where does NETGRAPH live?
- How does NETGRAPH interface with FreeBSD user space?

- Working with NETGRAPH
  - Nodes and Hooks
  - Control Messages
  - The `ngctl(8)` command in action
  - Visualizing NETGRAPH systems

- Details of frequently used NETGRAPH nodes
  - `ng_ether, ng_eiface`
  - `ng_socket, ng_ksocket`
  - `ng_bridge, ng_vlan, ng_tee`
  - `ng_etf, ng_one2many`

- Examples of how to use NETGRAPH nodes as building blocks
  - Interface (snooping) example
  - Ethernet filter example
  - Interface bonding example

- Investigating more sophisticated examples
  - IP Content filtering using `ng_ipfw`, `ng_tag`, and `ng_bpf`
  - WAN Bridge using `ng_bridge`, `ng_eiface`, `ng_ksocket`, and OpenVPN or IPSEC
  - `mpd5:` NETGRAPH based implementation of the multi-link PPP protocol for FreeBSD

- Guidelines for implementing one's own NETGRAPH node
  - Understanding `mbuf`'s
  - Essential node type methods
  - `ng_sample`
  - Debugging a NETGRAPH type

A question and answer session concludes the tutorial.

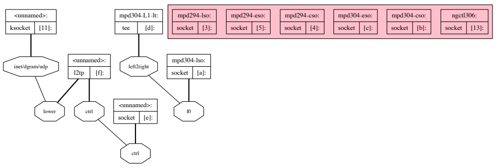
Lecturer Biography

Adrian Steinmann earned a Ph.D. in Mathematical Physics from Swiss Federal Institute of Technology in Zürich, Switzerland, and has over 20 years experience as an IT consultant and software developer. He is founder of Webgroup Consulting AG, a Swiss consulting company.

He has been working with FreeBSD since 1993 and became NetBSD committer in December 2011. He develops and maintains the STYX system based on BSD to offer remote managed services and to build custom systems on small x86 based platforms. This enabling technology has also been used to build secure encryption appliances on commodity hardware for the Swiss IT industry.

He is fluent in Perl, C, English, German, Italian, and has passion and flair for finding straightforward solutions to intricate problems.

During his free time he likes to play Go, to hike, and to sculpt.



Introduction to NETGRAPH on FreeBSD Systems

Dr. Adrian Steinmann <ast@marabu.ch>

AsiaBSDCon 2012

Tokyo University of Science, Tokyo, Japan
22 – 25 March, 2012

Tutorial Materials

Extended Abstract

Tutorial Slides

‘All About Netgraph’

man 8 ngctl

man 8 nghook

man 4 netgraph

About Myself

- Ph.D. in Mathematical Physics (long time ago)
- Webgroup Consulting AG (now)
- IT Consulting: Open Source, Security, Perl
- FreeBSD since version 1.0 (1993)
- NetBSD since version 3.0 (2005)
- Traveling, Sculpting, Go

Tutorial Outline

(1) Netgraph in a historical context

(2) Getting to know netgraph

(3) Working with netgraph

(4) Details of frequently used netgraph node types

(5) Examples using netgraph nodes as building blocks

(6) Investigate some more sophisticated examples

(7) Guidelines for implementing custom node types

What is Netgraph

Netgraph is in-kernel ‘network plumbing’

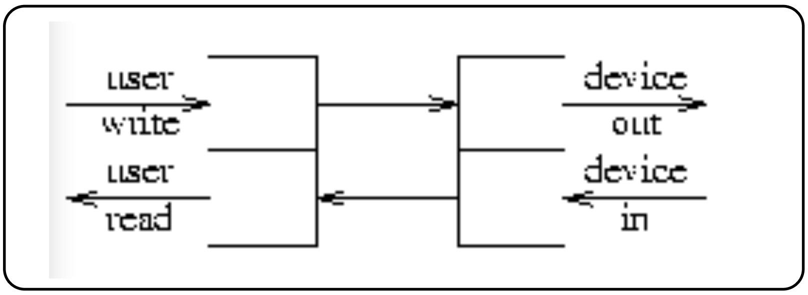
Drawn as a *graph*, a communication protocol can be seen as data packets flowing (bidirectionally) along the **edges** (lines) between **nodes** where the packets are processed.

In FreeBSD since version 3.0 (2000)

Created on FreeBSD 2.2 (1996) by Archie Cobbs <archie@freebsd.org> and Julian Elischer <julian@freebsd.org> for the Whistle InterJet at Whistle Communications, Inc.

Ancient History

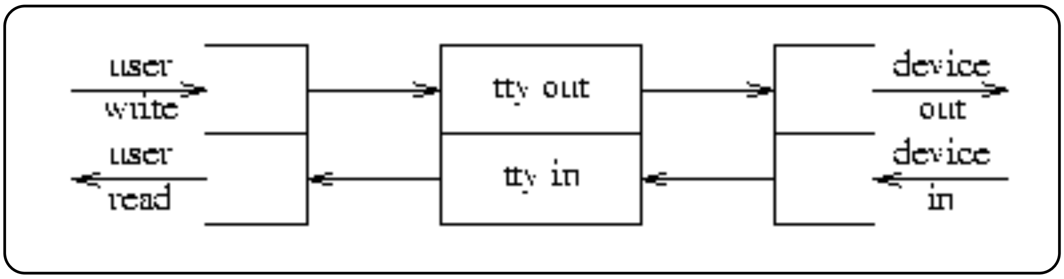
Dennis Ritchie (Eighth Edition Research Unix, Bell Labs)
October 1984: ‘A Stream Input-Output System’



After open() of a plain device

Ancient History

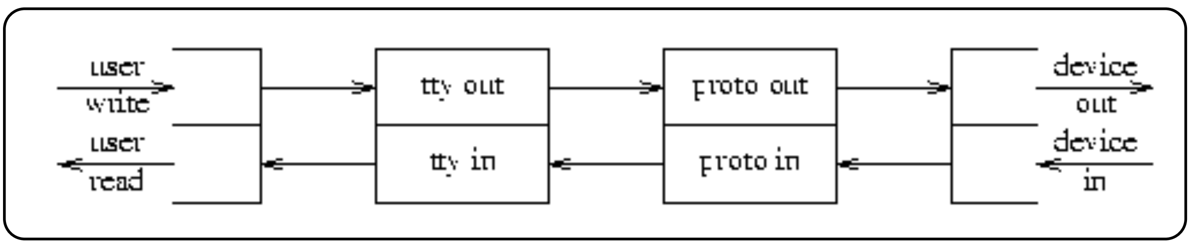
Dennis Ritchie (Eighth Edition Research Unix, Bell Labs)
October 1984: 'A Stream Input-Output System'



After open() of TTY device

Ancient History

Dennis Ritchie (Eighth Edition Research Unix, Bell Labs)
October 1984: 'A Stream Input-Output System'



Networked TTY device

System V STREAMS

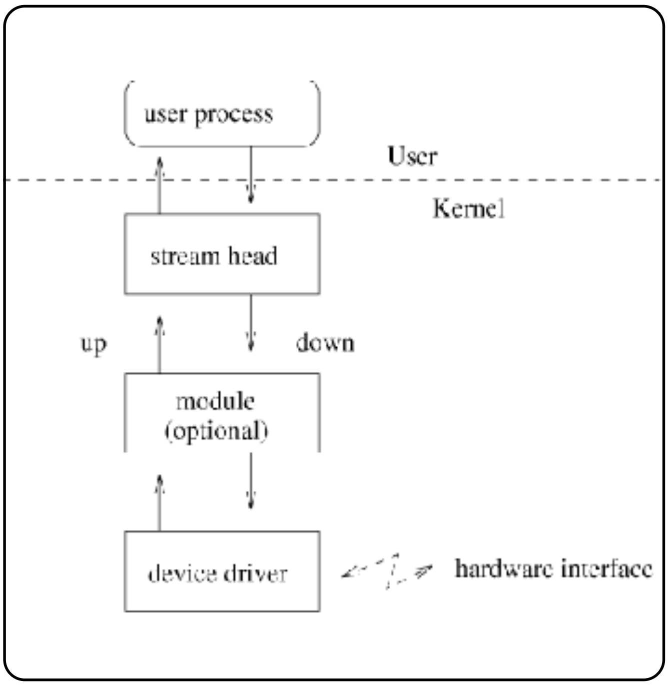
SVR3, SVR4, X/Open

Push modules onto a stack pointed to by the handle returned by the open() system call to the underlying device driver.

System V STREAMS

FreeBSD supports basic STREAMS system calls – see **`streams(4)`**

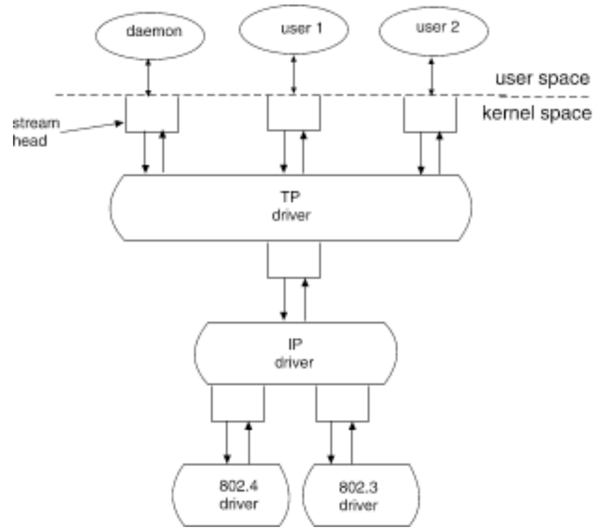
Linux STREAMS (LiS)



STREAMS Multiplexors

A transport protocol (TP) supporting multiple simultaneous STREAMS multiplexed over IP.

The TP routes data from the single lower STREAM to the appropriate upper STREAM.

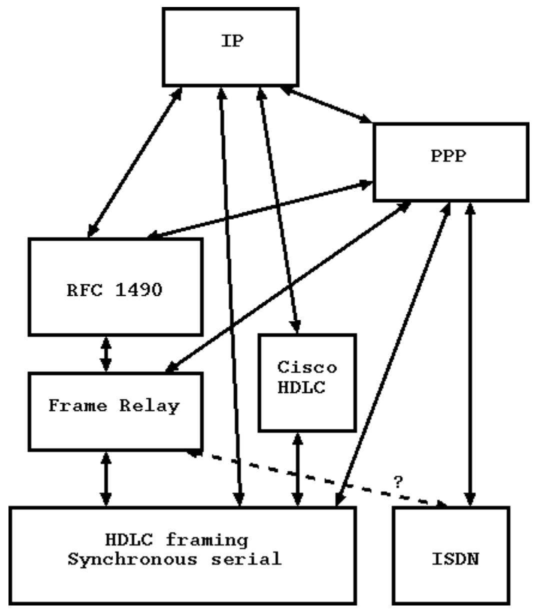


'All About Netgraph'

Best intro to NETGRAPH: 'All About Netgraph' by Archie Cobbs Daemon News, March 2000

http://people.freebsd.org/~julian/netgraph.html

(last retrieved February 2012)



Ways to talk IP over synchronous serial and ISDN WAN connections

Netgraph Platforms

- FreeBSD 2.2; mainline as of FreeBSD 3.0
- Port to NetBSD 1.5 (from FreeBSD 4.3) but not in mainline NetBSD
- DragonFly BSD "Netgraph7" (upgrade from netgraph from FreeBSD 4.x to FreeBSD 7.x)
- In one commercial Linux

  6WindGate's VNB (Virtual Networking Blocks) are derived from netgraph

New Approaches Today
Streamline (on Linux)

Herbert Bos
www.cs.vu.nl/~herbertb
VU University, Amsterdam

Scalable I/O Architecture (ACM TOCS 2011)

Address network latency problem on OS design level
Beltway Buffers, Ring Buffers: minimize copying in IPC
PipeFS for visualizing/manipulating nodes/graph

New Approaches Today
netmap (on FreeBSD)

Luigi Rizzo
http://info.iet.unipi.it/~luigi/
Università di Pisa, Italy

SIGCOMM 2011 Best Poster Award

Memory mapped access to network devices
Fast and safe network access for user space applications

Evolution on FreeBSD

- Netgraph continues to be in mainline FreeBSD tree since 3.0 (started with10 netgraph modules)
- New networking abstractions appearing in FreeBSD remain netgraph-aware via additional netgraph modules

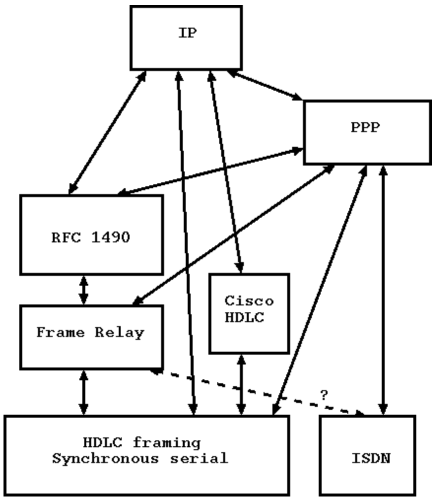
  (4.4: ~20, 4.9: ~25, 5.4: ~30, 6.3: ~45, 7.2:~50, 8.x: ~60)
- Netgraph in 5.x: added SMP (that is: locking, queuing, timers, bidirectional hooks)
- Netgraph in 8.x: support for VIMAGE
  most recent addition: ng_patch(4)

Timeline on FreeBSD

Initial “NETGRAPH 2”

FreeBSD 2.2 customized for Whistle InterJet

10 modules



Timeline on FreeBSD

First Public Release

FreeBSD 3.3

16 modules

`async cisco echo frame_relay hole iface ksocket lmi ppp pppoe rfc1490 socket tee tty UI vjc`

Timeline on FreeBSD

NETGRAPH

FreeBSD 4.11

16 + 2 modules

async cisco echo frame_relay hole iface ksocket lmi ppp pppoe rfc1490 socket tee tty UI vjc

bpf ether

Timeline on FreeBSD

NETGRAPH

FreeBSD 5.5 / FreeBSD 6.x

18 + ~30 modules

async cisco echo frame_relay hole iface ksocket lmi ppp pppoe rfc1490 socket tee tty UI vjc

bpf ether

atm atmllc atmpif **bluetooth bridge** bt3c btsocket device eiface etf fec **gif** gif_demux h4 hci hub ip_input l2cap **l2tp** mppc **netflow one2many** pptpgre split sppp sscfu sscop ubt uni **vlan**

Timeline on FreeBSD

"NETGRAPH 7"

FreeBSD 7.4 / 8.x

~48 + ~12 modules

`async cisco echo frame_relay hole iface ksocket lmi ppp pppoe rfc1490 socket tee tty UI vjc`

`bpf ether`

`atm atmllc atmpif bluetooth bridge bt3c btsocket device eiface etf fec gif gif_demux h4 hci hub ip_input l2cap l2tp mppc netflow one2many pptpgre split sppp sscfu sscop ubt uni vlan`

`car ccatm deflate` **`ipfw`** `nat` **`patch`** `pred1 source sync_ar sync_sr` **`tag`** `tcpmms`

Software Licensing

The netgraph framework is in FreeBSD kernel, hence it is under BSD license

Netgraph nodes may have any license

'A Gentlemen's Agreement – Assessing The GNU General Public License and its Adaptation to Linux' by Douglas A. Hass, Chicago-Kent Journal of Intellectual Property, 2007 mentioned netgraph as an example to be followed:

For example, FreeBSD uses a networking architecture called netgraph. Along with the rest of FreeBSD, this modular architecture accepts modules under virtually any license. Unlike Linux, Netgraph's API integrates tightly with the FreeBSD kernel, using a well-documented set of standard function calls, data structures, and memory management schemes. Regardless of the underlying licensing structure, modules written for netgraph compliance must interact with netgraph's structure in a predictable, predefined manner.

★ man 4 netgraph

The aim of netgraph is to supplement rather than replace the existing kernel networking infrastructure

- A flexible way of combining protocol and link level drivers
- A modular way to implement new protocols
- A common framework for kernel entities to inter-communicate
- A reasonably fast, kernel-based implementation

More

★ ‘All About Netgraph’

★ man -k netgraph

- netgraph(3) is the programmer’s API
- News archives:

  Many (sometimes good) questions, less answers, some working examples, and very often “I did that once but don’t have the working example here right now” ... (for some counterexamples, see URLS at end of tutorial slides)

How to Build a Netgraph

(i) Create a node *node_A* (with unconnected hooks)

(ii) Create a peer node *node_B*, connecting one of the hooks of *node_A* (*hook_A*) to one of the hooks of *node_B* (*hook_B*) – this creates an edge from *node_A* to *node_B* along *hook_A* to *hook_B*

(iii) Repeat step (i) or step (ii), or:

Connect an unconnected hook to another one, creating a new edge between existing nodes

Control Messages

- Nodes can receive control messages, they reply to them by setting a reply flag
- Control messages are C structures with a (generic) netgraph type cookie and variable payload
- A node can also define its own message types by taking a unique netgraph type cookie

Addressing Nodes

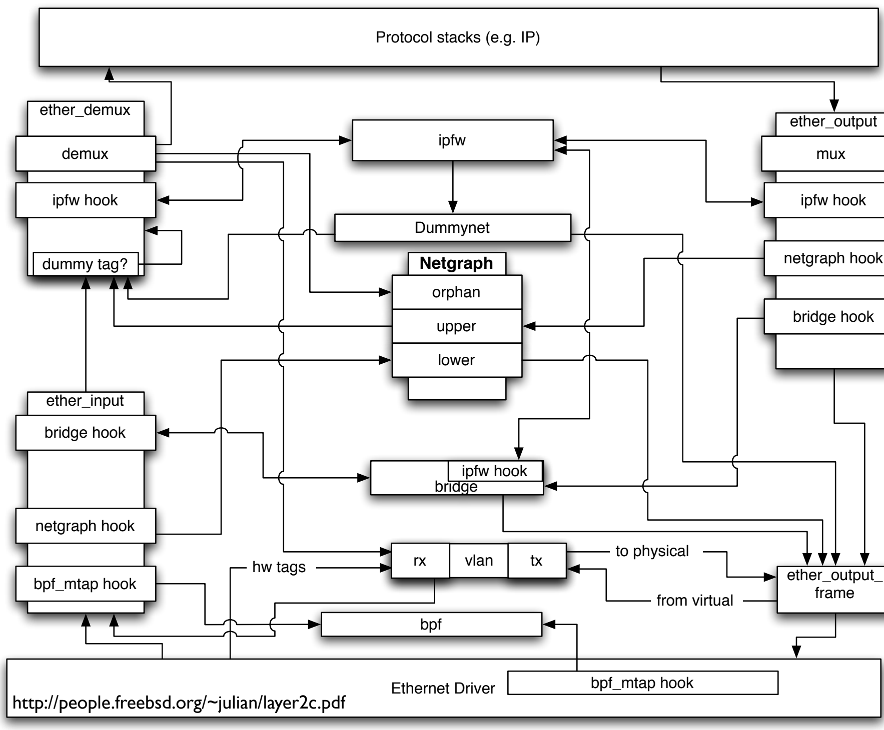
- Every node has an address (also called path) and an internal ID
- A named node can be addressed absolutely

  nodename: (for example, **`em0:`**)

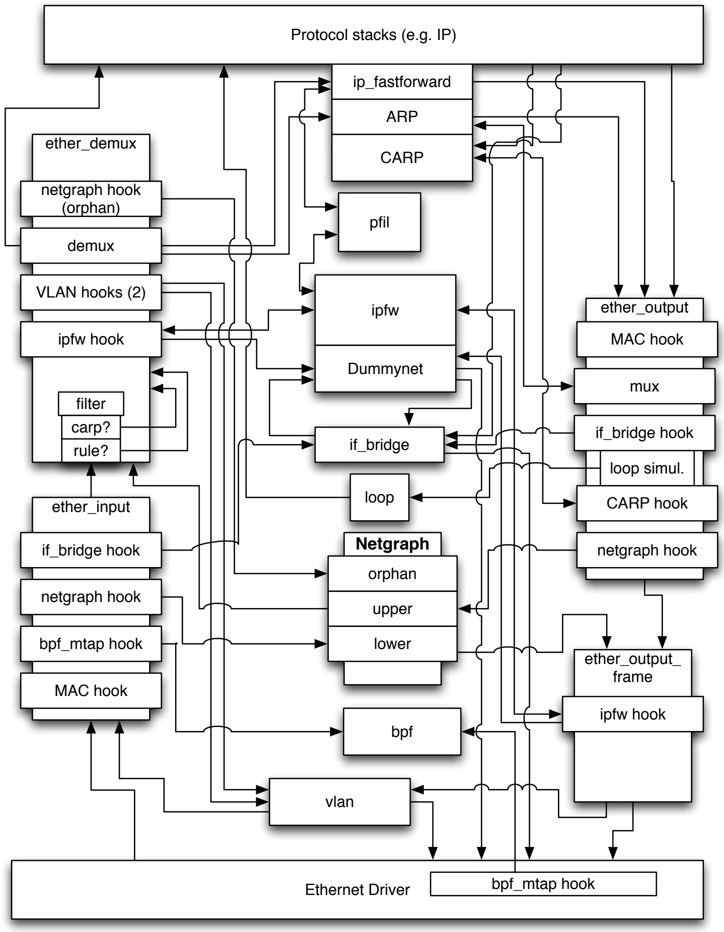
- A nameless node can be addressed absolutely via its internal ID

  If node has internal ID **`0000007e`**, it can be address as **`[7e]:`**

Where NETGRAPH hooks live in FreeBSD 4.x kernel



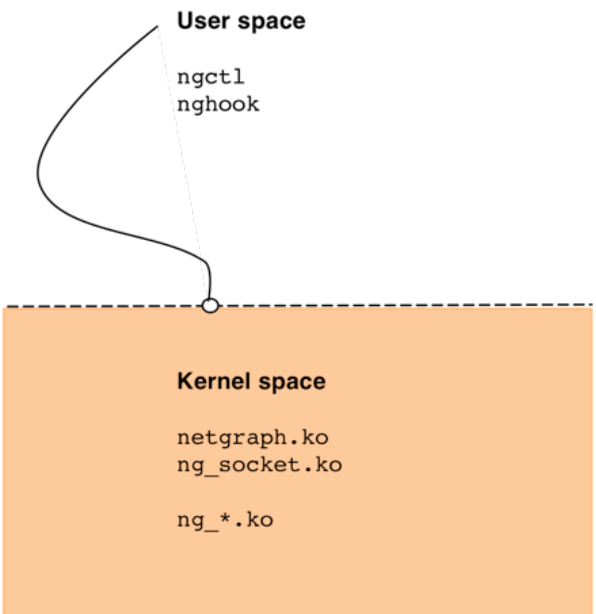
Where NETGRAPH hooks live in FreeBSD 6.x kernel



Netgraph User/Kernel Interface

BSD Sockets!

AF_NETGRAPH address family for ctrl, data protocols (netstat -f ... -p ...)



ngctl

```
# ngctl
Available commands:
  config      get or set configuration of node at <path>
  connect     Connects hook <peerhook> of the node at <relpath> ...
  debug       Get/set debugging verbosity level
  dot         Produce a GraphViz (.dot) of the entire netgraph.
  help        Show command summary or get more help on a specific ..
  list        Show information about all nodes
  mkpeer      Create and connect a new node to the node at "path"
  msg         Send a netgraph control message to the node at "path"
  name        Assign name <name> to the node at <path>
  read        Read and execute commands from a file
  rmhook      Disconnect hook "hook" of the node at "path"
  show        Show information about the node at <path>
  shutdown    Shutdown the node at <path>
  status      Get human readable status information from the node ...
  types       Show information about all installed node types
  write       Send a data packet down the hook named by "hook".
  quit        Exit program
+ ^D
```

Starting Netgraph

Netgraph automatically loads necessary kernel modules

```
# kldstat
Id Refs Address    Size     Name
 1    8 0xc0400000 9fad10   kernel
 2    1 0xc0dfb000 6a45c    acpi.ko

# ngctl list
There are 1 total nodes:
  Name: ngctl39213   Type: socket   ID: 00000001   Num hooks: 0

# kldstat
Id Refs Address    Size     Name
 1    8 0xc0400000 9fad10   kernel
 2    1 0xc0dfb000 6a45c    acpi.ko
 6    1 0xc85ba000 4000     ng_socket.ko
 7    1 0xcc648000 b000     netgraph.ko
```

Querying Netgraph Status

```
# ngctl &
# netstat -f ng
Netgraph sockets
Type   Recv-Q Send-Q Node Address    #Hooks
ctrl        0      0 ngctl1314:         0
data        0      0 ngctl1314:         0

# ngctl list
There are 1 total nodes:
  Name: ngctl99971   Type: socket   ID: 00000008   Num hooks: 0
```

There used to be a bug in 7.x, 8.x introduced 2006 (when ng_socket.c became a loadable module) which caused netstat -f netgraph to fail (see kern/140446)

Creating Nodes

- ng_ether and ng_gif nodes are created automatically once the corresponding kernel module is loaded (and once loaded, they cannot be unloaded)
- ngctl mkpeer <node> <ngtype> <hook> <peerhook>

  node is usually **`[id]:`** or **`name:`**

  – the default is the current node

  `ngctl mkpeer em3: tee lower right`

Naming Nodes

- ngctl name <target> <name>
  where <target> is node:hook

  `ngctl name em3:lower T`

- Example: hook *lower* is one of the three hooks for the ng_ether (*lower* connects to raw ether device, *upper* to the upper protocols, *orphans* is like *lower*, but only receives unrecognized packets – see man ng_ether)

  `ngctl mkpeer T: MyType right2left MyHookR2L`

  `ngctl name T:right2left MyNode`

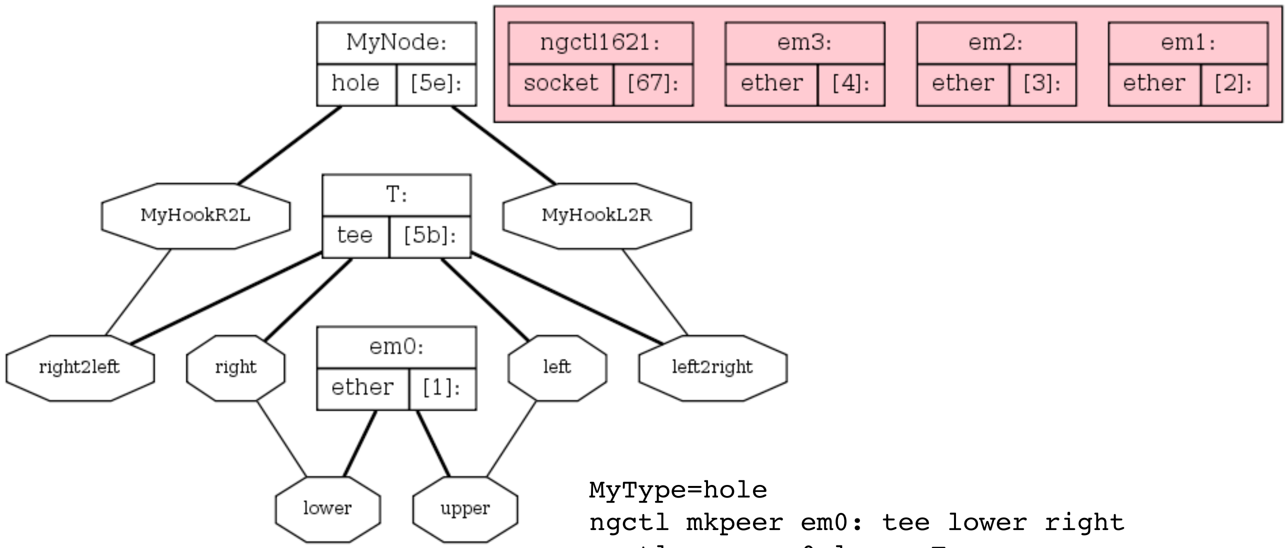
Connecting Nodes

- Connecting nodes creates edges
- Hooks that are not connected do nothing, i.e. packets that would go there according to node type are dropped
- ngctl connect is used when both nodes already exist (as opposed to mkpeer, where a new node is created and connected in one step)
- ngctl connect <node_a> <node_b> <hook_a> <hook_b>

`ngctl connect MyNode: T: MyHookL2R left2right`

`ngctl connect T: em0: left upper`

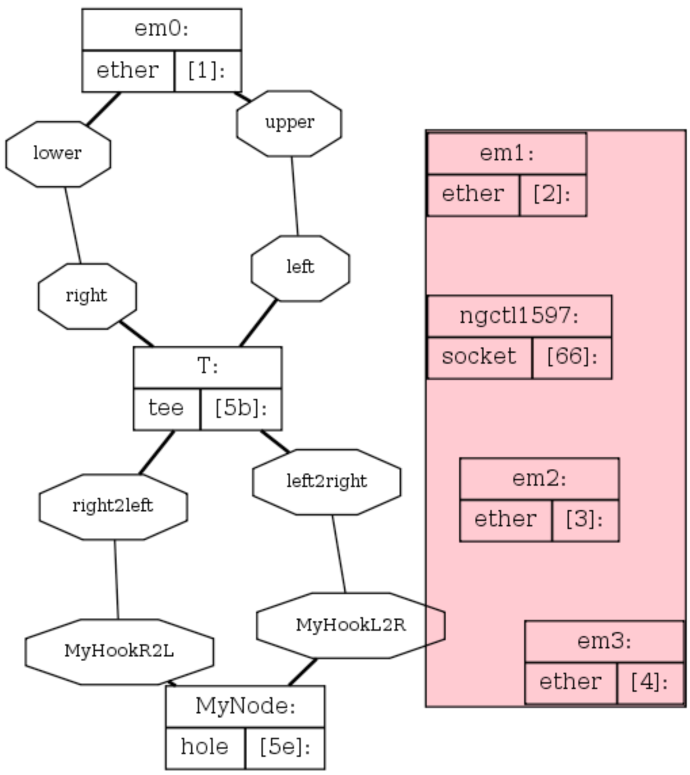
A first Net-graph




```
MyType=hole
ngctl mkpeer em0: tee lower right
ngctl name em0:lower T
ngctl mkpeer T: $MyType right2left MyHookR2L
ngctl name T:right2left MyNode
ngctl connect MyNode: T: MyHookL2R left2right
ngctl connect T: em0: left upper
```

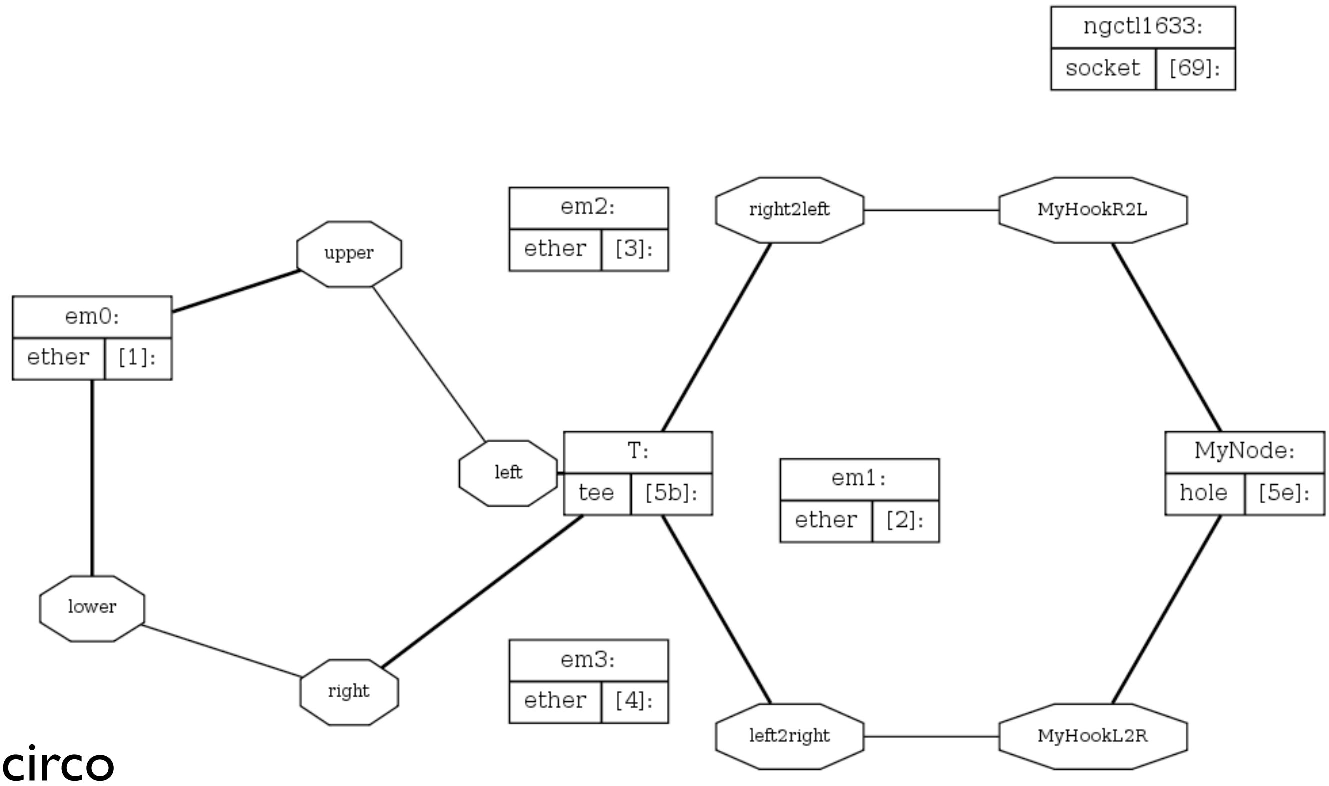
dot

Another Net-graph



neato

And another Net-graph



circo

Speaking with Nodes

- ngctl msg <node> <command> [args ...]

  `ngctl msg T: getstats`

- What messages a particular node accepts (and what arguments it takes) depends on the node – read section 4 of the manual for ng_xyz

- Netgraph has ascii syntax for passing binary information to node – see, for example, ng_ksocket(4) or ng_bpf(4)

Removing an Edge

- ngctl rmhook <hook>

  ```
  ngctl rmhook MyHookR2L
  ```

- Specifying the node is optional:

  ngctl rmhook <node> <hook>

  ```
  ngctl rmhook MyNode MyHookL2R
  ```

Removing a Node

- Shutting down a node (all edges to hooks on this node are removed)
- ngctl shutdown <node>

  ```
  ngctl shutdown T:
  ```

  These edges disappear:

  ```
  T:left - em0:lower
  T:right - em0:upper
  T:left2right - MyNode:MyHookL2R
  T:right2left - MyNode:MyHookL2R
  ```

Common Node Types

ng_ether(4) **`fxp0:`**

- Hooks: upper, lower, orphans
- Process raw ethernet traffic to/from other nodes
- Attach to actual 802.11 hardware and are named automatically
- Messages

getifname, getifindex, getenaddr, setenaddr,

getpromisc, setpromisc, getautosrc, setautosrc,

addmulti, delmulti, detach

Common Node Types

ng_eiface(4) **`ngeth0:`**

- Hooks: ether
- Virtual ethernet interface providing ethernet framing
- Messages:

  set, getifname

More Nodes Types

ng_iface(4) Virtual interface for protocol-specific frames

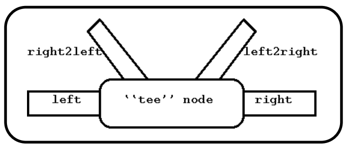
- Hooks: inet, inet6, ipx, atalk, ns, atm, natm
- Messages: getifname, point2point, broadcast, getipaddr, getifindex

`ng0:`

ng_tee(4) Useful for tapping (for example, to snoop traffic)

- Hooks: left, right, left2right, right2left
- Messages: getstats, clrstats, getclrstats

Example: Snooping




```
kldload ng_ether
ngctl mkpeer ${int}: tee lower left
ngctl name ${int}:lower T

ngctl connect ${int}: T: upper right
ngctl show T:
  Name: T              Type: tee              ID: 0000003e   Num hooks: 2
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  right           ${int}             ether        00000019        upper
  left            ${int}             ether        00000019        lower

nghook -an T: left2right
...
0020:  d0 aa e4 d2 14 84 47 ae 24 fb 40 fd df 9b 80 10  ......G.$.@.....

ngctl msg T: getstats
Rec'd response "getstats" (1) from "[66]:":
Args:   { right={ inOctets=15332 inFrames=117 outOctets=13325 outFrames=126 } left=
{ inOctets=13325 inFrames=126 outOctets=15332 outFrames=117 } left2right=
{ outOctets=1027 outFrames=6 } }

ngctl shut T:
ngctl shut ${int}:
```

Nodes of Special Interest

ng_socket(4)

- Enables the user-space manipulation (via a socket) of what is normally a kernel-space entity (the associated netgraph node)
- Hooks: arbitrary number with arbitrary names

ng_ksocket(4)

- Enables the kernel-space manipulation (via a netgraph node) of what is normally a user-space entity (the associated socket)
- Hooks: exactly one, name defines <family>/<type>/<proto>

These two examples show how to use ng_ksocket(4):

- **`/usr/share/examples/netgraph/ngctl`**
- **`/usr/share/examples/netgraph/udp.tunnel`**

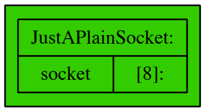
Visualizing Netgraph

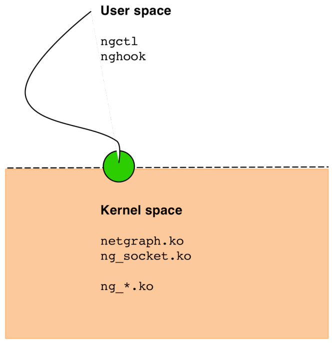
- Graph Visualization Software from AT&T and Bell Labs: http://www.graphviz.org/
- Port graphics/graphviz; various layouts:

dot	filter for hierarchical layouts of graphs
neato	filter for symmetric layouts of graphs
twopi	filter for radial layouts of graphs
circo	filter for circular layout of graphs
fdp	filter for symmetric layouts of graphs

Graphing Netgraph Graphs

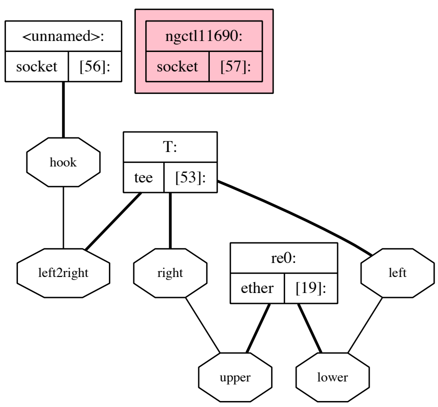
```
# ngctl
+ name [8]: JustAPlainSocket
+ dot
graph netgraph {
        edge [ weight = 1.0 ];
        node [ shape = record, fontsize = 12 ] {
                "8" [ label = "{JustAPlainSocket:|{socket|[8]:}}" ];
        };
        subgraph cluster_disconnected {
                bgcolor = pink;
                "8";
        };
};
^D
$ dot -Tpng ngctl.dot > ngctl.png
```





Snoop Example Revisited

```
int=re0
ngctl mkpeer ${int}: tee lower left
ngctl name ${int}:lower T
ngctl connect ${int}: T: upper right
nghook -an T: left2right >/dev/null &
ngctl dot
ngctl shut T:
ngctl shut ${int}:
```




```
graph netgraph {
        edge [ weight = 1.0 ];
        node [ shape = record, fontsize = 12 ] {
                "5e" [ label = "{\<unnamed\>:|{socket|[5e]:}}" ];
                "56" [ label = "{\<unnamed\>:|{socket|[56]:}}" ];
                "19" [ label = "{${int}:|{ether|[19]:}}" ];
                "5f" [ label = "{ngctl11738:|{socket|[5f]:}}" ];
                "5b" [ label = "{T:|{tee|[5b]:}}" ];
        };
        subgraph cluster_disconnected {
                bgcolor = pink;
                "56";
                "5f";
        };
        node [ shape = octagon, fontsize = 10 ] {
                "5e.hook" [ label = "hook" ];
        };
        {
                edge [ weight = 2.0, style = bold ];
                "5e" -- "5e.hook";
        };
        "5e.hook" -- "5b.left2right";
        node [ shape = octagon, fontsize = 10 ] {
                "19.upper" [ label = "upper" ];
                "19.lower" [ label = "lower" ];
        };
        {
                edge [ weight = 2.0, style = bold ];
                "19" -- "19.upper";
                "19" -- "19.lower";
        };
        node [ shape = octagon, fontsize = 10 ] {
                "5b.left2right" [ label = "left2right" ];
                "5b.right" [ label = "right" ];
                "5b.left" [ label = "left" ];
        };
        {
                edge [ weight = 2.0, style = bold ];
                "5b" -- "5b.left2right";
                "5b" -- "5b.right
                "5b" -- "5b.left";
        };
        "5b.right" -- "19.upper";
        "5b.left" -- "19.lower";
};
```

Bridge

ng_bridge(4)

Hooks:

link0, ..., link31 (NG_BRIDGE_MAX_LINKS = 32)

Messages:

setconfig, getconfig, reset, getstats, clrstats, getclrstats, gettable

This module does bridging on ethernet node types, each link carries raw ethernet frames so ng_bridge(4) can learn which MACs are on which link (and does loop detection). Typically, the link0 hook is connected to the first ng_ether(4) lower hook with ngctl mkpeer, and the subsequent ethernet interfaces' lower hooks are then connected with ngctl connect, making sure that the ethernet interfaces are in promiscuous mode and do not set the source IP:

```
ngctl msg ${int}: setpromisc 1 && ngctl msg ${int}: setautosrc 0
```

– see /usr/share/examples/netgraph/ether.bridge

One2Many

ng_one2many(4)

Hooks:

one, many1, many2, ...

Messages:

setconfig, getconfig, getstats, clrstats, getclrstats, gettable

Similar to ng_bridge() the one hook is connected to the first ng_ether(4) upper hook with ngctl mkpeer, and the subsequent ethernet interfaces' upper hooks are connected with ngctl connect, making sure that the ethernet interfaces are in promiscuous mode and do not set the source IP.

Example: Interface Bonding

```
kldload ng_ether
kldload ng_one2many

intA=vr0
intB=vr1
IP=192.168.1.1/24

# start with clean slate
ngctl shut ${intA}:
ngctl shut ${intB}:

# Plumb nodes together
ngctl mkpeer ${intA}: one2many upper one
ngctl connect ${intA}: ${intA}:upper lower many0
ngctl connect ${intB}: ${intA}:upper lower many1

# Allow ${intB} to xmit/recv ${intA} frames
ngctl msg ${intB}: setpromisc 1
ngctl msg ${intB}: setautosrc 0

# Configure all links as up
ngctl msg ${intA}:upper setconfig "{ xmitAlg=1 failAlg=1 enabledLinks=[ 1 1 ] }"

# Bring up interface
ifconfig ${intA} ${IP}

ngctl msg ${intA}:lower getconfig
Rec'd response "getconfig" (1) from "[11]:":
Args:   { xmitAlg=1 failAlg=1 enabledLinks=[ 1 1 ] }
```

VLAN

ng_vlan(4)

Hooks: downstream, nomatch, <arbitrary_name>

Messages: addfilter, delfilter, gettable

```
ETHER_IF=vr0

kldload ng_ether
kldload ng_vlan

ngctl shutdown ${ETHER_IF}:
ngctl mkpeer ${ETHER_IF}: vlan lower downstream
ngctl name ${ETHER_IF}:lower vlan
ngctl connect ${ETHER_IF}: vlan: upper nomatch

ngctl mkpeer vlan: eiface vlan123 ether
ngctl msg vlan: addfilter '{ vlan=123 hook="vlan123" }'

ngctl show vlan:
  Name: vlan            Type: vlan            ID: 00000038   Num hooks: 3
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  vlan123         <unnamed>       eiface       0000003c        ether
  nomatch         vr0             ether        0000000b        upper
  downstream      vr0             ether        0000000b        lower
```

Ethernet Filter

ng_etf(4)

Hooks:

downstream, nomatch, <any_name>

Messages:

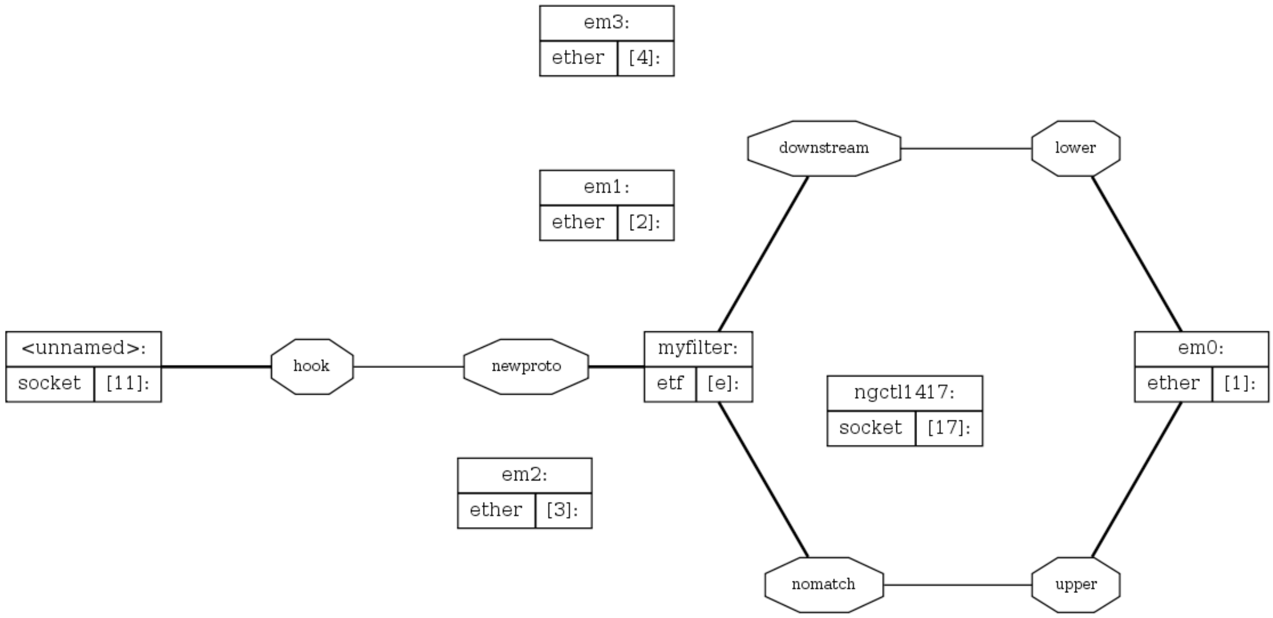
getstatus, setfilter

The downstream hook is usually connected to an ng_ether lower hook and the nomatch hook to the corresponding ng_ether upper; the <any_name> hook can then be captured by, say, an nghook process

Example: ng_etf(4)

```
kldload ng_ether
kldload ng_etf

int=fxp0

MATCH1=0x834
MATCH2=0x835

# Clean up any old connection, add etf node and name it
ngctl shutdown ${int}:lower
ngctl mkpeer ${int}: etf lower downstream
ngctl name ${int}:lower myfilter

# Connect the nomatch hook to the upper part of the same interface.
# All unmatched packets will act as if the filter is not present.
ngctl connect ${int}: myfilter: upper nomatch

# Show packets on a new hook "newproto" in background, ignore stdin
nghook -an myfilter: newproto &

# Filter the two random ethertypes to this newhook
ngctl msg myfilter: setfilter "{ matchhook=\"newproto\" ethertype=${MATCH1} }"
ngctl msg myfilter: setfilter "{ matchhook=\"newproto\" ethertype=${MATCH2} }"
```

Example: ng_etf(4)



Berkeley Packet Filter

ng_bpf(4)

Hooks: an arbitrary number of <arbitrary_name> hooks

Messages: setprogram, getprogram,

getstats, clrstats, getclrstats

The setprogram message expects a BPF(4) filter program **`bpf_prog_len=16 bpf_prog=[...]`** which is generated from tcpdump -ddd output. Together with this filter one sets the **`thisHook="inhook"`**, **`ifMatch="matchhook"`**, and **`ifNotMatch="notmatchhook"`** hooks for use elsewhere.

Generating BPF Program

```
#!/bin/sh
i=0
{
    tcpdump -s 8192 -ddd "$@" \
        | while read line; do
            set -$- $line
            i=$(( $i + 1 ))
            if [ $i -eq 1 ]
            then
                echo "bpf_prog_len=$1"
                continue
            elif [ $i -eq 2 ]; then
                echo "bpf_prog=["
            fi
            echo " { code=$1 jt=$2 jf=$3 k=$4 }"
        done
    echo " ]"
} | xargs
exit 0
```

```
tcpdump -s 8192 -ddd \
     tcp dst port 80

16
40 0 0 12
21 0 4 34525
48 0 0 20
21 0 11 6
40 0 0 56
21 8 9 80
21 0 8 2048
48 0 0 23
21 0 6 6
40 0 0 20
69 4 0 8191
177 0 0 14
72 0 0 16
21 0 1 80
6 0 0 8192
6 0 0 0
```

```
tcpdump2bpf.sh tcp dst port 80
bpf_prog_len=16 bpf_prog=[ { code=40 jt=0 jf=0 k=12 } { code=21 jt=0 jf=4 k=34525 }
{ code=48 jt=0 jf=0 k=20 } ...  { code=6 jt=0 jf=0 k=8192 } { code=6 jt=0 jf=0 k=0 } ]
```

ng_ipfw(4), ng_tag(4)

- ng_ipfw(4) interface between IPFW and Netgraph

  Hooks: arbitrary number, name must be numeric

  Messages: none (only generic)

- ng_tag(4)

  Hooks: arbitrary number and name

  Messages: sethookin, gethookin, sethookout, gethookout, getstats, clrstats, getclrstats

BPF + IPFW + TAG = L7 Filter

RTFM ng_tag(4)

DirectConnect P2P TCP payloads contain the string "$Send|", filter these out with ng_bpf(4), tag the packets with ng_tag(4), and then block them with an ipfw rule hooking them to a ng_ipfw(4) node.

man ng_tag(4)

```
kldload ng_ipfw; kldload ng_bpf; kldload ng_tag

ngctl mkpeer ipfw: bpf 41 ipfw
ngctl name ipfw:41 dcbpf
ngctl mkpeer dcbpf: tag matched th1
ngctl name dcbpf:matched ngdc

grep MTAG_IPFW /usr/include/netinet/ip_fw.h
#define MTAG_IPFW 1148380143  /* IPFW-tagged cookie */

ngctl msg ngdc: sethookin  { thisHook=\"th1\" ifNotMatch=\"th1\" }
ngctl msg ngdc: sethookout { thisHook=\"th1\" \
                             tag_cookie=1148380143 tag_id=412 }
ngctl msg dcbpf: setprogram { thisHook=\"matched\" \
                              ifMatch=\"ipfw\" bpf_prog_len=1 \
                              bpf_prog=[ { code=6 k=8192 } ] }

tcpdump2bpf.sh "ether[40:2]=0x244c && ether[42:4]=0x6f636b20"
bpf_prog_len=6 bpf_prog=[ {... { code=6 jt=0 jf=0 k=8192 } ... ]

ipfw add 100 netgraph 41 tcp from any to any iplen 46
ipfw add 110 reset tcp from any to any tagged 412

sysctl net.inet.ip.fw.one_pass=0
```

"$Lock "

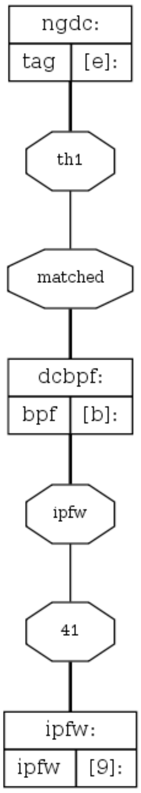
BPF + IPFW + TAG = L7 Filter

DirectConnect P2P TCP payloads contain the string "$Send|", filter these out with ng_bpf(4), tag the packets with ng_tag(4), and then block them with an ipfw rule hooking them to a ng_ipfw(4) node.

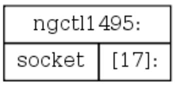
man ng_tag(4)
http://www.ipp2p.org/ipt_ipp2p.c

```
ipfw add 200 allow ip from any to any
kldload ng_ipfw; kldload ng_bpf; kldload ng_tag

ngctl mkpeer ipfw: bpf 41 ipfw
ngctl name ipfw:41 dcbpf
ngctl mkpeer dcbpf: tag matched th1
ngctl name dcbpf:matched ngdc

grep MTAG_IPFW /usr/include/netinet/ip_var.h
#define MTAG_IPFW 1148380143  /* IPFW-tagged cookie */

ngctl msg ngdc: sethookin  { thisHook=\"th1\" ifNotMatch=\"th1\" }
ngctl msg ngdc: sethookout { thisHook=\"th1\" \
                             tag_cookie=1148380143 tag_id=412 }
ngctl msg dcbpf: setprogram { thisHook=\"matched\" \
                              ifMatch=\"ipfw\" bpf_prog_len=6 \
                              bpf_prog=[ { ... (see below) ... } ] }

tcpdump2bpf.sh "ether[40:2]=0x2453 && ether[42:4]=0x656e647c"
bpf_prog_len=6 bpf_prog=[ bpf_prog_len=6 bpf_prog=[ { code=40 jt=0 jf=0 k=40 }
{ code=21 jt=0 jf=3 k=9299 } { code=32 jt=0 jf=0 k=42 } { code=21 jt=0 jf=1
k=1701733500 } { code=6 jt=0 jf=0 k=8192 } { code=6 jt=0 jf=0 k=0 } ]

ipfw add 100 netgraph 41 tcp from any to any iplen 46
ipfw add 110 reset tcp from any to any tagged 412

sysctl net.inet.ip.fw.one_pass=0
```

BPF + IPFW + TAG = L7 Filter




```
ngctl mkpeer ipfw: bpf 41 ipfw
ngctl name ipfw:41 dcbpf
ngctl mkpeer dcbpf: tag matched th1
ngctl name dcbpf:matched ngdc
ngctl msg ngdc: sethookin  { thisHook=\"th1\" ifNotMatch=\"th1\" }
ngctl msg ngdc: sethookout { thisHook=\"th1\" \
                           tag_cookie=1148380143 tag_id=412 }
ngctl msg dcbpf: setprogram { thisHook=\"matched\" \
                            ifMatch=\"ipfw\" bpf_prog_len=6 \
                            bpf_prog=[ { ... } ] }
```




```
ipfw add 100 netgraph 41 tcp from any to any iplen 46
ipfw add 110 reset tcp from any to any tagged 412

sysctl net.inet.ip.fw.one_pass=0
```

MPD Champion of netgraph(3) User Library

```
ng_async
ng_bpf
ng_car
ng_deflate
ng_ether
ng_iface
ng_ksocket
ng_l2tp
ng_mppc
ng_nat
ng_netflow
ng_ppp
ng_pppoe
ng_pptpgre
ng_pred1
ng_socket
ng_tcpmss
ng_tee
ng_tty
ng_vjc
```

http://sourceforge.net/projects/mpd/files/

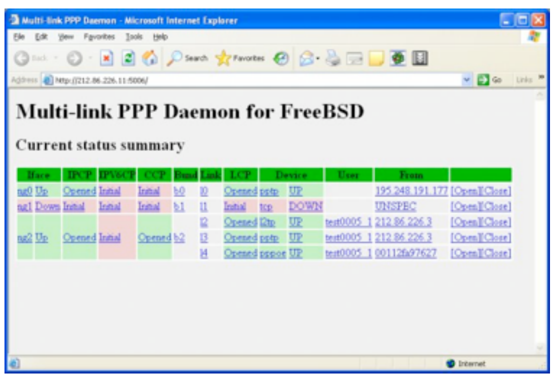
"MPD is a netgraph based PPP implementation for FreeBSD. MPD5 supports thousands of Sync, Async, PPTP, L2TP, PPPoE, TCP and UDP links in client, server and access concentrator (LAC/PAC/TSA) modes. It is very fast and functional"

Uses netgraph User Library (libnetgraph, -lnetgraph)

man 3 netgraph

Scales well

Built-in web monitor



MPD Multi-link PPP Daemon Link Types

- **modem** to connect using different asychronous serial connections, including modems, ISDN terminal adapters, and null-modem. Mpd includes event-driven scripting language for modem identification, setup, manual server login, etc.
- **pptp** to connect over the Internet using the Point-to-Point Tunnelling Protocol (PPTP). This protocol is supported by the most OSes and hardware vendors
- **l2tp** to connect over the Internet using the Layer Two Tunnelling Protocol (L2TP). L2TP is a PPTP successor supported with modern clients and servers
- **pppoe** to connect over an Ethernet port using the PPP-over-Ethernet (PPPoE) protocol. This protocol is often used by DSL providers
- **tcp** to tunnel PPP session over a TCP connection. Frames are encoded in the same was as asychronous serial connections
- **udp** to tunnel PPP session over a UDP connection. Each frame is encapsulated in a UDP datagram packet
- **ng** to connect to netgraph nodes

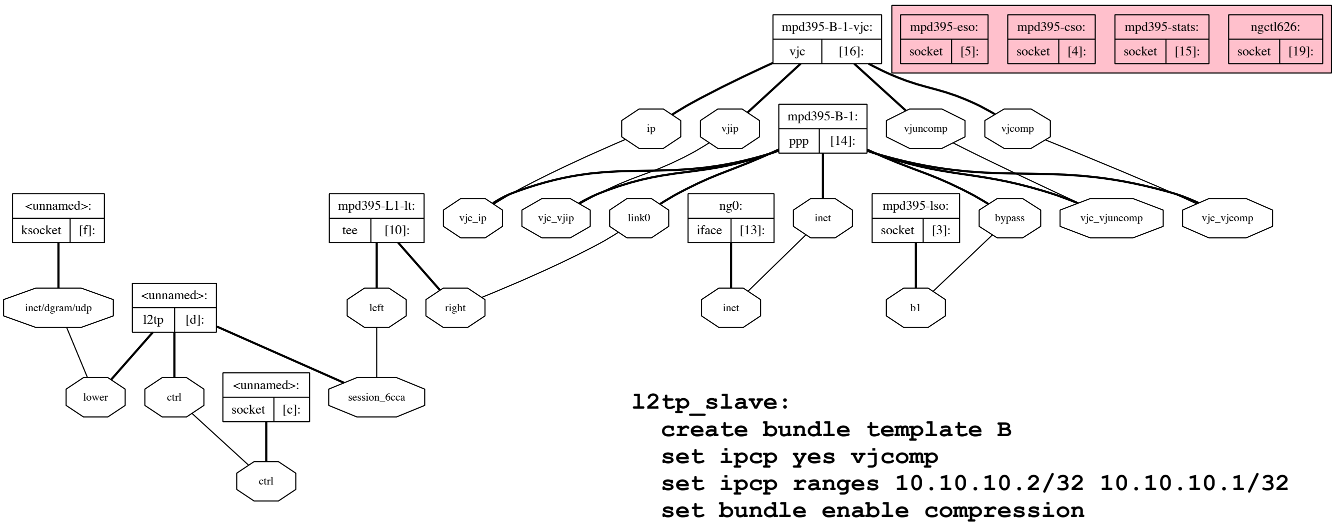
MPD Multi-link PPP Daemon Configuration

- MPD operates on several protocol layers (OSI layering) and acts as a PPP terminator with Radius or other AAA and IP accounting; usually used for IP but could be used for other protocols
- Can act in PPP repeater mode too (L2TP or PPTP access concentrator)
- PPP Terminator Layers:

  Interface – NCPs – Compression/Encryption – Bundle – Links

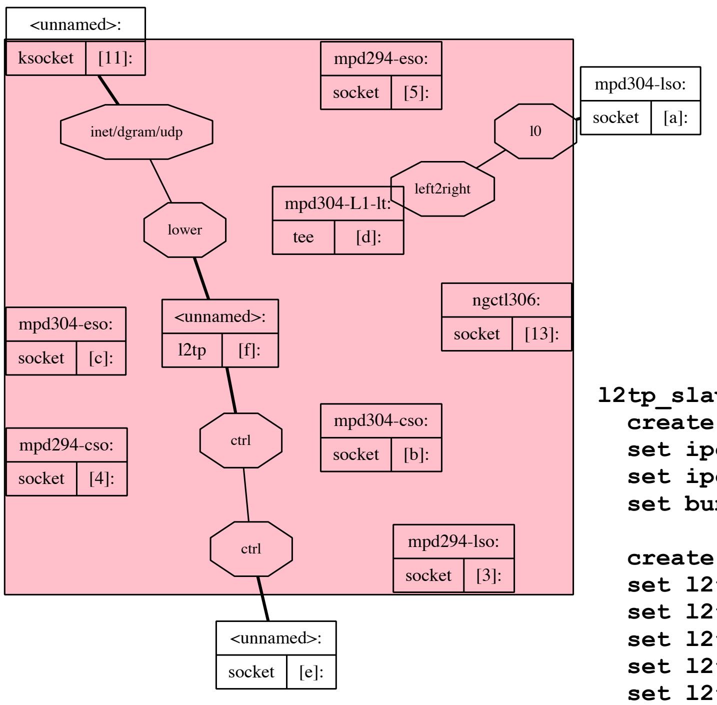
  A set of Links is a Bundle connecting to the peer, after optional compression/encryption the NCP (IPCP or IPv6CP) layer presents the interface which is visible via the ifconfig command

MPD NETGRAPH Net Graph Running




```
l2tp_slave:
  create bundle template B
  set ipcp yes vjcomp
  set ipcp ranges 10.10.10.2/32 10.10.10.1/32
  set bundle enable compression

  create link static L1 l2tp
  set l2tp self 10.10.10.11 1701
  set l2tp peer 10.10.10.10 1701
  set l2tp enable outcall
  set l2tp hostname slave
  set l2tp secret MySecret
  set link max-redial 0
  set link action bundle B
```

MPD NETGRAPH Net Graph Stopped




```
l2tp_slave:
  create bundle template B
  set ipcp yes vjcomp
  set ipcp ranges 10.10.10.2/32 10.10.10.1/32
  set bundle enable compression

  create link static L1 l2tp
  set l2tp self 10.10.10.11 1701
  set l2tp peer 10.10.10.10 1701
  set l2tp enable outcall
  set l2tp hostname slave
  set l2tp secret MySecret
  set link max-redial 0
  set link action bundle B
```

Creating Custom Node Type

- “Only” two steps:

  (1) Define your new custom **`struct ng_type`**

  (2) **`NETGRAPH_INIT(tee, &ng_tee_typestruct)`**

- **`sys/netgraph/ng_tee.c`** is a good example
- **`netgraph.h`**

  Defines basic netgraph structures

- **`ng_message.h`**

  Defines structures and macros for control messages, here you see how the generic control messages are implemented

Custom Node ng_sample.c

Skeleton module in sys/netgraph/ng_sample.{c,h}

```
static struct ng_type typestruct = {
        .version =      NG_ABI_VERSION,
        .name =         NG_XXX_NODE_TYPE,
        .constructor =  ng_xxx_constructor,
        .rcvmsg =       ng_xxx_rcvmsg,
        .shutdown =     ng_xxx_shutdown,
        .newhook =      ng_xxx_newhook,
/*      .findhook =     ng_xxx_findhook,       */
        .connect =      ng_xxx_connect,
        .rcvdata =      ng_xxx_rcvdata,
        .disconnect =   ng_xxx_disconnect,
        .cmdlist =      ng_xxx_cmdlist,
};
```

About mbuf(9)

An mbuf is a basic unit of memory management in kernel. Network packets and socket buffers use mbufs. A network packet may span multiple mbufs arranged into a linked list, which allows adding or trimming headers with minimal overhead.

Netgraph must have `M_PKTHDR` flag set, i.e., `struct pkthdr m_pkthdr` is added to the mbuf header. This means you also have access and are responsible for the data packet header information.

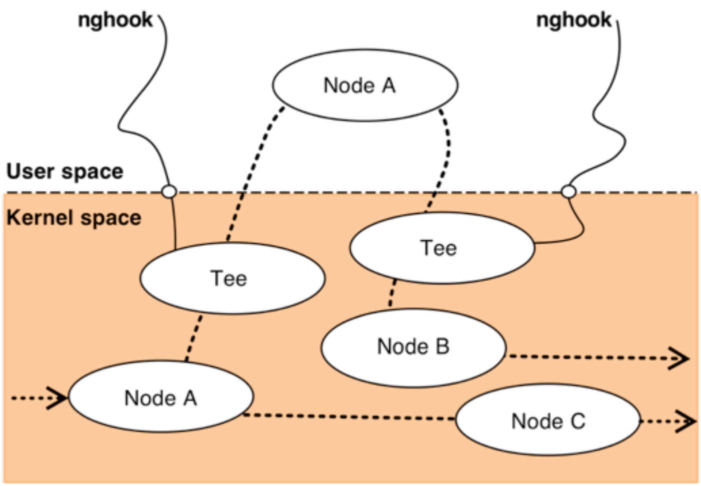
`m_pullup(mbuf, len)` is expensive, so always check if is needed:

```
struct foobar *f;

if (m->m_len < sizeof(*f) && (m = m_pullup(m, sizeof(*f))) == NULL) {
    NG_FREE_META(meta);
    return (ENOBUFS);
}
f=mtod(m, struct foobar *);
...
```

User-space Prototyping

- User library libnetgraph – netgraph(3) – has a sufficiently similar API to the netgraph(4) kernel API, so that node types can be prototyped in user space

- Use an intermediate ng_tee(4) node and attach nghook(8) for debugging



Q & A

(1) Netgraph in a historical context

(2) Getting to know netgraph

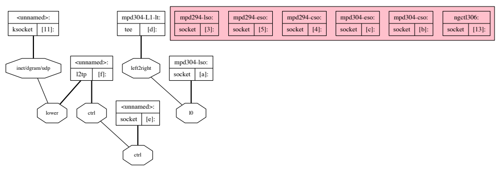
(3) Working with netgraph

(4) Details of frequently used netgraph node types

(5) Examples using netgraph nodes as building blocks

(6) Investigate some more sophisticated examples

(7) Guidelines for implementing custom node types



THANK YOU
for attending the
Introduction to NETGRAPH
on FreeBSD Systems
Tutorial

URLS

(all URLS last retrieved in February 2012)

'All About Netgraph' (complete introduction to FreeBSD netgraph)

http://people.freebsd.org/~julian/netgraph.html

'Netgraph in 5 and beyond'

A BAFUG talk where Julian Elischer points out things he fixed as FreeBSD transition from 4.x to 5.x (slides and movie)

http://people.freebsd.org/~julian/BAFUG/talks/Netgraph/

Re: diagram of 4.10 layer 2 spaghetti

http://www.mail-archive.com/freebsd-net@freebsd.org/msg16970.html

http://people.freebsd.org/~julian/layer2c.pdf

'Netgraph7' on DragonFly BSD

http://leaf.dragonflybsd.org/mailarchive/submit/2011-02/msg00008.html

http://gitweb.dragonflybsd.org/~nant/dragonfly.git/shortlog/refs/heads/netgraph7

Debugging a netgraph node

http://lists.freebsd.org/pipermail/freebsd-net/2009-June/022292.html

URLS, URLS

(all URLS last retrieved in February 2012)

STREAMS

http://cm.bell-labs.com/cm/cs/who/dmr/st.html (Initial article by Dennis Ritchie)
http://www.linuxjournal.com/article/3086 (LiS: Linux STREAMS)
http://en.wikipedia.org/wiki/STREAMS

"Hacking" The Whistle InterJet © (i486 internet access appliance 1996)

http://www.anastrophe.com/~paul/wco/interjet/ (Information on the Whistle Interjet)

6WINDGate™ Linux

Closed source virtual network blocks (VNB) stack derived from netgraph technology (press release 2007)
http://www.windriver.com/partner-validation/1B-2%20%20-%206WINDGate%20Architecture%20Overview%20v1.0.pdf

(ACM TOCS'11) Application-tailored I/O with Streamline

http://www.cs.vu.nl/~herbertb/papers/howard_ndss11.pdf

Marko Zec – network emulation using the virtualized network stack in FreeBSD

http://www.imunes.net/virtnet/eurobsdcon07_tutorial.pdf (Network stack virtualization for FreeBSD 7.0, EuroBSDCon Sept 2007)
http://meetbsd.org/files/2_05_zec.pdf (Network emulation using the virtualized network stack in FreeBSD, MeetBSD July 2010)
http://www.youtube.com/watch?v=wh09MirPd5Y (MeetBSD talk, July 2010)

URLS, URLS, URLS

(all URLS last retrieved in February 2012)

FreeBSD CVS Repository (to see all the currently available netgraph modules)
http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netgraph/
Latest addition ng_patch(4) 2010 by Maxim Ignatenko:
http://www.mail-archive.com/freebsd-net@freebsd.org/msg32164.html

A Gentlemen's Agreement
Assessing The GNU General Public License and its Adaptation to Linux
by Douglas A. Hass, Chicago-Kent Journal of Intellectual Property, 2007 mentions that netgraph has a good license model
http://papers.ssrn.com/sol3/papers.cfm?abstract_id=951842

Subject: NetBSD port of the freebsd netgraph environment
NetBSD 1.5, that is (port from the FreeBSD 4.3 netgraph)
http://mail-index.netbsd.org/tech-net/2001/08/17/0000.html

Netgraph on Debian GNU / kFreeBSD (in russian)
http://morbow.blogspot.com/2011/02/netgraph-debian.html

URLS, URLS, URLS, URLS

(all URLS last retrieved in February 2012)

Subject: [PATCH] ng_tag - new netgraph node, please test (L7 filtering possibility)
“Yes, netgraph always was a semi-programmer system”
Includes an example of in-kernel L7 (bittorrent) pattern matching filter using ng_bpf, ng_mtag and ipfw
http://lists.freebsd.org/pipermail/freebsd-net/2006-June/010898.html

(SIGCOMM Poster 2011 Winner) netmap: fast and safe access to network for user programs
http://info.iet.unipi.it/~luigi/netmap/20110815-sigcomm-poster.pdf
http://info.iet.unipi.it/~luigi/netmap/rizzo-ancs.pdf

Re: option directive and turning on AOE
In a discussion on ATA over Ethernet (frame type 0x88a2) other questions came up:
Does netgraph have locking issues? Is netgraph performant? – it depends:
http://unix.derkeiler.com/Mailing-Lists/FreeBSD/arch/2004-09/0006.html

Reprint of

'All About Netgraph'

by Archie Cobbs
<archie@freebsd.org>

Published in
Daemon News
March 2000

http://people.freebsd.org/~julian/netgraph.html

All About Netgraph

By Archie Cobbs <archie@freebsd.org>

Part I: What is Netgraph?

The motivation

Imagine the following scenario: you are developing a TCP/IP router product based on FreeBSD. The product needs to support bit-synchronous serial WAN connections, i.e., dedicated high speed lines that run up to T1 speeds, where the basic framing is done via HDLC. You need to support the following protocols for the transmission of IP packets over the wire:

- IP frames delivered over HDLC (the simplest way to transmit IP)
- IP frames delivered over ``Cisco HDLC'' (basically, packets are prepended with a two-byte Ethertype, and there are also periodic keep-alive packets).
- IP delivered over frame relay (frame relay provides for up to 1000 virtual point-to-point links which are multiplexed over a single physical wire).
- IP inside RFC 1490 encapsulation over frame relay (RFC 1490 is a way to multiplex multiple protocols over a single connection, and is often used in conjunction with frame relay).
- Point-to-Point Protocol (PPP) over HDLC
- PPP over frame relay
- PPP inside RFC 1490 encapsulation over frame relay
- PPP over ISDN
- There are even rumors you might have to support frame relay over ISDN (!)

Figure 1 graphically indicates all of the possible combinations:

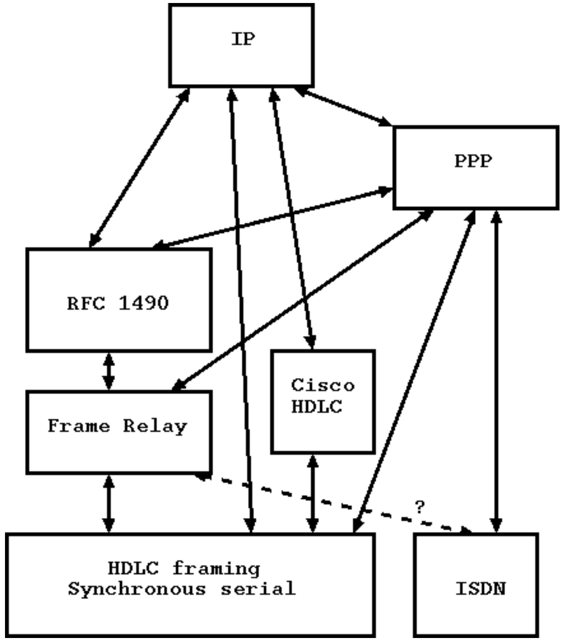


Figure 1: Ways to talk IP over synchronous serial and ISDN WAN connections

This was the situation faced by Julian Elischer <julian@freebsd.org> and myself back in 1996 while we were working on the Whistle InterJet. At that time FreeBSD had very limited support for synchronous serial hardware and protocols. We looked at OEMing from Emerging Technologies, but decided instead to do it ourselves.

The answer was **netgraph**. Netgraph is an in-kernel networking subsystem that follows the UNIX principle of achieving power and flexibility through combinations of simple tools, each of which is designed to perform a single, well defined task. The basic idea is straightforward: there are **nodes** (the tools) and **edges** that connect pairs of nodes (hence the ``graph'' in ``netgraph''). **Data packets** flow bidirectionally along the edges from node to node. When a node receives a data packet, it performs some processing on it, and then (usually) forwards it to another node. The processing may be something as simple as adding/removing headers, or it may be more complicated or involve other parts of the system. Netgraph is vaguely similar to System V Streams, but is designed for better speed and more flexibility.

Netgraph has proven very useful for networking, and is currently used in the

Whistle InterJet for all of the above protocol configurations (except frame relay over ISDN), plus normal PPP over asynchronous serial (i.e., modems and TAs) and Point-to-Point Tunneling Protocol (PPTP), which includes encryption. With all of these protocols, the data packets are handled entirely in the kernel. In the case of PPP, the negotiation packets are handled separately in user-mode (see the FreeBSD port for mpd-3.0b5).

Nodes and edges

Looking at the picture above, it is obvious what the nodes and edges might be. Less obvious is the fact that a node may have an arbitrary number of connections to other nodes. For example, it is entirely possible to have both IP, IPX, and PPP running inside RFC 1490 encapsulation at the same time; indeed, multiplexing multiple protocols is exactly what RFC 1490 is for. In this case, there would be three edges connecting into the RFC 1490 node, one for each protocol stack. There is no requirement that data flow in any particular direction or that a node have any limits on what it can do with a data packet. A node can be a source/sink for data, e.g., associated with a piece of hardware, or it can just modify data by adding/removing headers, multiplexing, etc.

Netgraph nodes live in the kernel and are semi-permanent. Typically, a node will continue to exist until it is no longer connected to any other nodes. However, some nodes are **persistent**, e.g., nodes associated with a piece of hardware; when the number of edges goes to zero typically the hardware is shutdown. Since they live in the kernel, nodes are not associated with any particular process.

Control messages

This picture is still oversimplified. In real life, a node may need to be configured, queried for its status, etc. For example, PPP is a complicated protocol with lots of options. For this kind of thing netgraph defines **control messages**. A control message is ``out of band data.'' Instead of flowing from node to node like data packets, control messages are sent asynchronously and directly from one node to another. The two nodes don't have to be (even indirectly) connected. To allow for this, netgraph provides a simple **addressing scheme** by which nodes can be identified using simple ASCII strings.

Control messages are simply C structures with a fixed header (a `struct ng_mesg`) and a variable length payload. There are some control messages that all nodes understand; these are called the **generic control messages** and are implemented in the base system. For example, a node can be told to destroy itself or to make or break an edge. Nodes can also define their own type-specific control messages. Each node type that defines its own control messages must have a unique **typecookie**. The combination of the typecookie and **command** fields in the control message header determine how to interpret it.

Control messages often elicit responses in the form of a **reply control message**. For example, to query a node's status or statistics you might send the node a ``get status'' control message; it then sends you back a response (with the identifying **token** copied from the original request) containing the requested information in the payload area. The response control message header is usually identical to the original header, but with the **reply flag** set.

Netgraph provides a way to convert these structures to and from ASCII strings, making human interaction easier.

Hooks

In netgraph, edges don't really exist *per se*. Instead, an edge is simply an association of two **hooks**, one from each node. A node's hooks define how that node can be connected. Each hook has a unique, statically defined name that often indicates what the purpose of the hook is. The name is significant only in the context of that node; two nodes may have similarly named hooks.

For example, consider the Cisco HDLC node. Cisco HDLC is a very simple protocol multiplexing scheme whereby each frame is prepended with its Ethertype before transmission over the wire. Cisco HDLC supports simultaneous transmission of IP, IPX, AppleTalk, etc. Accordingly, the netgraph Cisco HDLC node (see `ng_cisco(8)`) defines hooks named **`inet`**, **`atalk`**, and **`ipx`**. These hooks are intended to connect to the corresponding upper layer protocol engines. It also defines a hook named **`downstream`** which connects to the lower layer, e.g., the node associated with a synchronous serial card. Packets received on `inet`, `atalk`, and `ipx` have the appropriate two byte header prepended, and then are forwarded out the `downstream` hook. Conversely, packets received on `downstream` have the header stripped off, and are forwarded out the appropriate protocol hook. The node also handles the periodic ``tickle'' and query packets defined by the Cisco HDLC protocol.

Hooks are always either connected or disconnected; the operation of connecting or disconnecting a pair of hooks is atomic. When a data packet is sent out a hook, if that hook is disconnected, the data packet is discarded.

Some examples of node types

Some node types are fairly obvious, such as Cisco HDLC. Others are less obvious but provide for some interesting functionality, for example the ability to talk directly to a device or open a socket from within the kernel.

Here are some examples of netgraph node types that are currently implemented in FreeBSD. All of these node types are documented in their corresponding man pages.

Echo node type: `ng_echo(8)`
: This node type accepts connections on any hook. Any data packets it receives are simply echoed back out the hook they came in on. Any non-generic control messages are likewise echoed back as replies.

Discard node type: `ng_disc(8)`
: This node type accepts connections on any hook. Any data packets and

control messages it receives are silently discarded.

Tee node type: **`ng_tee(8)`**

This node type is like a bidirectional version of the `tee(1)` utility. It makes a copy of all data passing through it in either direction (``right'' or ``left''), and is useful for debugging. Data packets arriving in ``right'' are sent out ``left'' and a copy is sent out ``right2left''; similarly for data packets going from ``left'' to ``right''. Packets received on ``right2left'' are sent out ``left'' and packets received on ``left2right'' are sent out ``right''.

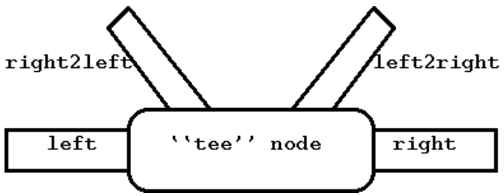


Figure 2: Tee node type

Interface node type: **`ng_iface(8)`**

This node type is both a netgraph node and a point-to-point system networking interface. It has (so far) three hooks, named ``inet'', ``atalk'', and ``ipx''. These hooks represent the protocol stacks for IP, AppleTalk, and IPX respectively. The first time you create an interface node, interface **ng0** shows up in the output of `ifconfig -a`. You can then configure the interface with addresses like any other point-to-point interface, ping the remote side, etc. Of course, the node must be connected to something or else your ping packets will go out the **inet** hook and disappear.

Unfortunately, FreeBSD currently cannot handle removing interfaces, so once you create an `ng_iface(8)` node, it remains persistent until the next reboot (however, this will be fixed soon).

```
$ ifconfig ng0 inet 1.1.1.1 2.2.2.2
```

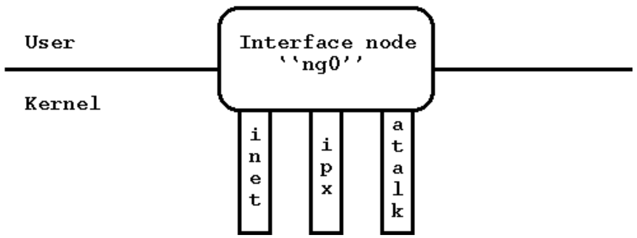


Figure 3: Interface node type

TTY node type: **`ng_tty(8)`**

This node type is both a netgraph node and an asynchronous serial line discipline (see `tty(4)`). You create the node by installing the `NETGRAPHDISC` line discipline on a serial line. The node has one hook called ``hook''. Packets received on ``hook'' are transmitted (as serial bytes) out the corresponding serial device; data received on the device is wrapped up into a packet and sent out ``hook''. Normal reads and writes to the serial device are disabled.

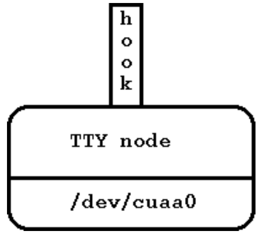


Figure 4: TTY node type

Socket node type: **`ng_socket(8)`**

This node type is very important, because it allows user-mode programs to participate in the netgraph system. Each node is both a netgraph node and a pair of sockets in the family `PF_NETGRAPH`. The node is created when a user-mode program creates the corresponding sockets via the `socket(2)` system call. One socket is used for transmitting and receiving netgraph data packets, while the other is used for control messages. The node supports hooks with arbitrary names, e.g. ``hook1,'' ``hook2,'' etc.

```
s1 = socket(PF_NETGRAPH, SOCK_DGRAM, NG_CONTRO
s2 = socket(PF_NETGRAPH, SOCK_DGRAM, NG_DATA);
```

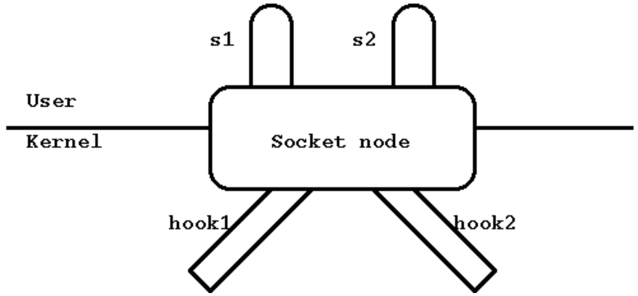


Figure 5: Socket node type

BPF node type: `ng_bpf(8)`
: This node type performs `bpf(4)` pattern matching and filtering on packets as they flow through it.

Ksocket node type: `ng_ksocket(8)`
: This node type is the reverse of `ng_socket(8)`. Each node is both a node and a socket that is completely contained in the kernel. Data received by the node is written to the socket, and vice-versa. The normal `bind(2)`, `connect(2)`, etc. operations are effected instead using control messages. This node type is useful for tunneling netgraph data packets within a socket connection (for example, tunneling IP over UDP).

Ethernet node type: `ng_ether(8)`
: If you compiled your kernel with `options NETGRAPH`, then every Ethernet interface is also a netgraph node with the same name as the interface. Each Ethernet node has two hooks, ``orphans'' and ``divert''; only one hook may be connected at a time. If ``orphans'' is connected, the device continues to work normally, except that all received Ethernet packets that have an unknown or unsupported Ethertype are delivered out that hook (normally these frames would simply be discarded). When the ``divert'' hook is connected, then *all* incoming packets are delivered out this hook. Packets received on either of these hooks are transmitted on the wire. All packets are raw Ethernet frames with the standard 14 byte header (but no checksum). This node type is used, for example, for PPP over Ethernet (PPPoE).

Synchronous drivers: `ar(4)` and `sr(4)`
: If you compiled your kernel with `options NETGRAPH`, the `ar(4)` and `sr(4)` drivers will have their normal functionality disabled and instead will operate as simple persistent netgraph nodes (with the same name as the device itself). Raw HDLC frames can be read from and written to the ``rawdata'' hook.

Meta information

In some cases, a data packet may have associated *meta-information* which needs to be carried along with the packet. Though rarely used so far, netgraph provides a mechanism to do this. An example of meta-information is priority information: some packets may have higher priority than others. Node types may define their own type-specific meta-information, and netgraph defines a `struct ng_meta` for this purpose. Meta-information is treated as opaque information by the base netgraph system.

Addressing netgraph nodes

Every netgraph node is addressable via an ASCII string called a **node address** or **path**. Node addresses are used exclusively for sending control messages.

Many nodes have **names**. For example, a node associated with a device will typically give itself the same name as the device. When a node has a name, it can always be addressed using the **absolute address** consisting of the name followed by a colon. For example, if you create an interface node named ``ng0'' it's address will be ``ng0:''.

If a node does not have a name, you can construct one from the node's unique **ID number** by enclosing the number in square brackets (every node has a unique ID number). So if node `ng0:` has ID number 1234, then ``[1234]:'' is also a valid address for that node.

Finally, the address ``.:'' or ``.'' always refers to the local (source) node.

Relative addressing is also possible in netgraph when two nodes are indirectly connected. A relative address uses the names of consecutive hooks on the path from the source node to the target node. Consider this picture:

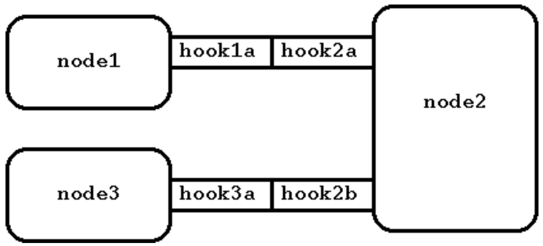


Figure 6: Sample node configuration

If **node1** wants to send a control message to **node2**, it can use the address ``.:hook1a'' or simply ``hook1a''. To address **node3**, it could use the address ``.:hook1a.hook2b'' or just ``hook1a.hook2b''. Conversely, **node3** could address **node1** using the address ``.:hook3a.hook2a'' or just ``hook3a.hook2a''.

Relative and absolute addressing can be combined, e.g., ``node1:hook1a.hook2b'' refers to **node3**.

Part II: Using Netgraph

Netgraph comes with command line utilities and a user library that allow interaction with the kernel netgraph system. Root privileges are required in order to perform netgraph operations from user-land.

From the command line

There are two command line utilities for interacting with netgraph, `nghook(8)` and `ngctl(8)`. `nghook(8)` is fairly simple: it connects to any unconnected hook of any existing node and lets you transmit and receive data packets via standard input and standard output. The output can optionally be decoded into human readable hex/ASCII format. On the command line you supply the node's absolute address and the hook name.

For example, if your kernel was compiled with `options NETGRAPH` and you have an Ethernet interface `fxp0`, this command will redirect all packets received by the Ethernet card and dump them to standard output in hex/ASCII format:

```
nghook -a fxp0: divert
```

The `ngctl(8)` is a more elaborate program that allows you to do most things possible in netgraph from the command line. It works in batch or interactive mode, and supports several commands that do interesting work, among them:

connect	Connects a pair of hooks to join two nodes
list	List all nodes in the system
mkpeer	Create and connect a new node to an existing node
msg	Send an ASCII formatted message to a node
name	Assign a name to a node
rmhook	Disconnect two hooks that are joined together
show	Show information about a node
shutdown	Remove/reset a node, breaking all connections
status	Get human readable status from a node
types	Show all currently installed node types
quit	Exit program

These commands can be combined into a script that does something useful. For example, suppose you have two private networks that are separated but both connected to the Internet via an address translating FreeBSD machine. Network A has internal address range 192.168.1.0/24 and external IP address 1.1.1.1, while network B has internal address range 192.168.2.0/24 and external IP address 2.2.2.2. Using netgraph you can easily set up a UDP tunnel for IP traffic between your two private networks. Here is a simple script that would do this (this script is also found in `/usr/share/examples/netgraph`):

```
#!/bin/sh

# This script sets up a virtual point-to-point WAN link between
# two subnets, using UDP packets as the ``WAN connection.''
# The two subnets might be non-routable addresses behind a
# firewall.
#

# Here define the local and remote inside networks as well
# as the local and remote outside IP addresses and the UDP
# port number that will be used for the tunnel.
#
LOC_INTERIOR_IP=192.168.1.1
LOC_EXTERIOR_IP=1.1.1.1
REM_INTERIOR_IP=192.168.2.1
REM_EXTERIOR_IP=2.2.2.2
REM_INSIDE_NET=192.168.2.0
UDP_TUNNEL_PORT=4028

# Create the interface node ``ng0'' if it doesn't exist already,
# otherwise just make sure it's not connected to anything.
#
if ifconfig ng0 >/dev/null 2>&1; then
        ifconfig ng0 inet down delete >/dev/null 2>&1

        ngctl shutdown ng0:
else
        ngctl mkpeer iface dummy inet
fi

# Attach a UDP socket to the ``inet'' hook of the interface node
# using the ng_ksocket(8) node type.
#
ngctl mkpeer ng0: ksocket inet inet/dgram/udp

# Bind the UDP socket to the local external IP address and port
#
ngctl msg ng0:inet bind inet/${LOC_EXTERIOR_IP}:${UDP_TUNNEL_PORT}

# Connect the UDP socket to the peer's external IP address and port
#
ngctl msg ng0:inet connect inet/${REM_EXTERIOR_IP}:${UDP_TUNNEL_PORT}

# Configure the point-to-point interface
#
ifconfig ng0 ${LOC_INTERIOR_IP} ${REM_INTERIOR_IP}

# Add a route to the peer's interior network via the tunnel
#
route add ${REM_INSIDE_NET} ${REM_INTERIOR_IP}
```

Here is an example of playing around with `ngctl(8)` in interactive mode. User input is shown in blue.

Start up `ngctl` in interactive mode. It lists the available commands...

```
$ ngctl
Available commands:
  connect    Connects hook <peerhook> of the node at <relpath> to <hook>
  debug      Get/set debugging verbosity level
  help       Show command summary or get more help on a specific command
  list       Show information about all nodes
  mkpeer     Create and connect a new node to the node at "path"
  msg        Send a netgraph control message to the node at "path"
  name       Assign name <name> to the node at <path>
  read       Read and execute commands from a file
  rmhook     Disconnect hook "hook" of the node at "path"
  show       Show information about the node at <path>
  shutdown   Shutdown the node at <path>
  status     Get human readable status information from the node at <path>
  types      Show information about all installed node types
  quit       Exit program
```

ngctl creates a `ng_socket(8)` type node when it starts. This is our local netgraph node which is used to interact with other nodes in the system. Let's look at it. We see that it has a name ``ngctl652'' assigned to it by `ngctl`, it is of type ``socket,'' it has ID number 45, and has zero connected hooks, i.e., it's not connected to any other nodes...

```
+ show .
```

```
  Name: ngctl652        Type: socket          ID: 00000045   Num hooks: 0
```

Now we will create and attach a ``tee'' node to our local node. We will connect the ``right'' hook of the tee node to a hook named ``myhook'' on our local node. We can use any name for our hook that we want to, as `ng_socket(8)` nodes support arbitrarily named hooks. After doing this, we inspect our local node again to see that it has an unnamed ``tee'' neighbor...

```
+ help mkpeer
Usage:    mkpeer [path] <type> <hook> <peerhook>
Summary:  Create and connect a new node to the node at "path"
Description:
  The mkpeer command atomically creates a new node of type "type"
  and connects it to the node at "path". The hooks used for the
  connection are "hook" on the original node and "peerhook" on
  the new node. If "path" is omitted then "." is assumed.
+ mkpeer . tee myhook right
+ show .
  Name: ngctl652        Type: socket          ID: 00000045   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  myhook          <unnamed>       tee          00000046        right
```

Similarly, if we check the tee node, we see that it has our local node as it's neighbor connected to the ``right'' hook. The ``tee'' node is still an unnamed node. However we could always refer to it using the absolute address ``[46]:'' or the relative addresses ``.:myhook'' or ``myhook''...

```
+ show .:myhook
  Name: <unnamed>       Type: tee             ID: 00000046   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  right           ngctl652        socket       00000045        myhook
```

Now let's assign our tee node a name and make sure that we can refer to it that way...

```
+ name .:myhook mytee
+ show mytee:
  Name: mytee           Type: tee             ID: 00000046   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  right           ngctl652        socket       00000045        myhook
```

Now let's connect a Cisco HDLC node to the other side of the ``tee'' node and inspect the ``tee'' node again. We are connecting to the ``downstream'' hook of the Cisco HDLC node, so it will act like the tee node is the WAN connection. The Cisco HDLC is to the ``left'' of the tee node while our local node is to the ``right'' of the tee node...

```
+ mkpeer mytee: cisco left downstream
+ show mytee:
  Name: mytee           Type: tee             ID: 00000046   Num hooks: 2
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  left            <unnamed>       cisco        00000047        downstream
  right           ngctl652        socket       00000045        myhook
+
Rec'd data packet on hook "myhook":
0000:  8f 00 80 35 00 00 00 02 00 00 00 00 00 00 00 00  ...5............
0010:  ff ff 00 20 8c 08 40 00                          ... ..@.
+
Rec'd data packet on hook "myhook":
0000:  8f 00 80 35 00 00 00 02 00 00 00 00 00 00 00 00  ...5............
0010:  ff ff 00 20 b3 18 00 17                          ... ....
```

Hey, what's that?! It looks like we received some data packets on our ``myhook'' hook. The Cisco node is generating periodic keep-alive packets every 10 seconds. These packets are passing through the tee node (from ``left'' to ``right'') and ending up being received on ``myhook'', where `ngctl` is displaying them on the console.

Now let's take inventory of all the nodes currently in the system. Note that our two Ethernet interfaces show up as well, because they are persistent nodes and we compiled our kernel with `options NETGRAPH`...

```
+ list
There are 5 total nodes:
  Name: <unnamed>       Type: cisco           ID: 00000047   Num hooks: 1
  Name: mytee           Type: tee             ID: 00000046   Num hooks: 2
  Name: ngctl652        Type: socket          ID: 00000045   Num hooks: 1
  Name: fxp1            Type: ether           ID: 00000002   Num hooks: 0
  Name: fxp0            Type: ether           ID: 00000001   Num hooks: 0
+
Rec'd data packet on hook "myhook":
0000:  8f 00 80 35 00 00 00 02 00 00 00 00 00 00 00 00  ...5............
0010:  ff ff 00 22 4d 40 40 00                          ..."M@@.
```

OK, let's shutdown (i.e., delete) the Cisco HDLC node so we'll stop receiving that data...

```
+ shutdown mytee:left
+ show mytee:
  Name: mytee           Type: tee             ID: 00000046   Num hooks: 1
  Local hook      Peer name       Peer type    Peer ID         Peer hook
  ----------      ---------       ---------    -------         ---------
  right           ngctl652        socket       00000045        myhook
```

Now let's get the statistics from the tee node. Here we send it a control message and it sends back an immediate reply. The command and reply are converted to/from ASCII automatically for us by ngctl, as control messages are binary structures...

```
+ help msg
Usage:    msg path command [args ... ]
Aliases:  cmd
Summary:  Send a netgraph control message to the node at "path"
Description:
  The msg command constructs a netgraph control message from the
  command name and ASCII arguments (if any) and sends that
  message to the node.  It does this by first asking the node to
  convert the ASCII message into binary format, and re-sending the
  result. The typecookie used for the message is assumed to be
  the typecookie corresponding to the target node's type.
+ msg mytee: getstats
Rec'd response "getstats" (1) from "mytee:":
Args:   { right={ outOctets=72 outFrames=3 } left={ inOctets=72 inFrames=3 }
  left2right={ outOctets=72 outFrames=3 } }
```

The reply is simply an ASCII version of the `struct ng_tee_stats` returned in the *control message reply (this structure is defined in ng_tee.h). We see that three frames (and 72 octets) passed through the tee node from left to right. Each frame was duplicated and passed out the ``left2right'' hook (but since this hook was not connected those duplicates were dropped).*

OK, now let's play with a `ng_ksocket(8)` node...

```
+ mkpeer ksocket myhook2 inet/stream/tcp
+ msg .:myhook2 connect inet/127.0.0.1:13
ngctl: send msg: Operation now in progress
Rec'd data packet on hook "myhook":
0000:  54 75 65 20 46 65 62 20 20 31 20 31 31 3a 30 32  Tue Feb  1 11:02
0010:  3a 32 38 20 32 30 30 30 0d 0a                    :28 2000..
```

Here we created a TCP socket in the kernel using a `ng_ksocket(8)` node and connected it to the ``daytime'' service on the local machine, which spits out the current time. How did we know we could use ``inet/127.0.0.1:13'' as an argument to the ``connect'' command? It's documented in the `ng_ksocket(8)` man page.

OK, enough playing...

```
+ quit
```

libnetgraph(3)

There is also a user library `libnetgraph(3)` for use by netgraph programs. It supplies many useful routines which are documented in the man page. See the source code in `/usr/src/usr.sbin/ngctl` for an example of using it.

Part III: The Implementation

Functional nature

How is netgraph implemented? One of the main goals of netgraph is *speed*, which is why it runs entirely in the kernel. Another design decision is that netgraph is entirely functional. That is, no queuing is involved as packets traverse from node to node. Instead, direct function calls are used. Data packets are **packet header mbuf**'s, while meta-data and control messages are heap-allocated C structures (using malloc type `M_NETGRAPH`).

Object oriented nature

Netgraph is somewhat object-oriented in its design. Each **node type** is defined by an array of pointers to the **methods**, or C functions, that implement the specific behavior of nodes of that type. Each method may be left `NULL` to fall back to the default behavior.

Similarly, there are some control messages that are understood by all node types and which are handled by the base system (these are called **generic** control messages). Each node type may in addition define its own type-specific control messages. Control messages always contain a typecookie and a command, which together identify how to interpret the message. Each node type must define its own unique typecookie if it wishes to receive type-specific control messages. The generic control messages have a predefined typecookie.

Memory

Netgraph uses reference counting for node and hook structures. Each pointer to a node or a hook should count for one reference. If a node has a name, that also counts as a reference. All netgraph-related heap memory is allocated and free'd using malloc type `M_NETGRAPH`.

Synchronization

Running in the kernel requires attention to synchronization. Netgraph nodes normally run at **`splnet()`** (see `spl(9)`). For most node types, no special attention is necessary. Some nodes, however, interact with other parts of the kernel that run at different priority levels. For example, serial ports run at `spltty()` and so `ng_tty(8)` needs to deal with this. For these cases netgraph provides alternate data transmission routines that handle all the necessary queuing auto-magically (see ng_queue_data() below).

How to implement a node type

To implement a new node type, you only need to do two things:

1. Define a `struct ng_type`.
2. Link it in using the `NETGRAPH_INIT()` macro.

Step 2 is easy, so we'll focus on step 1. Here is `struct ng_type`, taken from netgraph.h:

```
/*
 * Structure of a node type
 */
struct ng_type {

    u_int32_t       version;        /* must equal NG_VERSION */
    const char      *name;          /* Unique type name */
    modeventhand_t  mod_event;      /* Module event handler (optional) */
    ng_constructor_t *constructor;  /* Node constructor */
    ng_rcvmsg_t     *rcvmsg;        /* control messages come here */
    ng_shutdown_t   *shutdown;      /* reset, and free resources */
    ng_newhook_t    *newhook;       /* first notification of new hook */
    ng_findhook_t   *findhook;      /* only if you have lots of hooks */
    ng_connect_t    *connect;       /* final notification of new hook */
    ng_rcvdata_t    *rcvdata;       /* date comes here */
    ng_rcvdata_t    *rcvdataq;      /* or here if being queued */
    ng_disconnect_t *disconnect;    /* notify on disconnect */

    const struct    ng_cmdlist *cmdlist;    /* commands we can convert */

    /* R/W data private to the base netgraph code DON'T TOUCH! */
    LIST_ENTRY(ng_type) types;              /* linked list of all types */
    int                 refs;               /* number of instances */
};
```

The `version` field should be equal to `NG_VERSION`. This is to prevent linking in incompatible types. The `name` is the unique node type name, e.g., ``tee''. The `mod_event` is an optional module event handler (for when the node type is loaded and unloaded) -- similar to a static initializer in C++ or Java.

Next are the **node type methods**, described in detail below. The `cmdlist` provides (optional) information for converting control messages to/from ASCII (see below), and the rest is private to the base netgraph code.

Node type methods

Each node type must implement these methods, defined in its `struct ng_type`. Each method has a default implementation, which is used if the node type doesn't define one.

`int constructor(node_p *node);`

Purpose: Initialize a new node by calling `ng_make_node_common()` and setting `node->private` if appropriate. Per-node initialization and memory allocation should happen here. `ng_make_node_common()` should be called first; it creates the node and sets the reference count to one.

Default action: Just calls `ng_make_node_common()`.

When to override: If you require node-specific initialization or resource allocation.

```
int rcvmsg(node_p node, struct ng_mesg *msg,
    const char *retaddr, struct ng_mesg **resp);
```

Purpose: Receive and handle a control message. The address of the sender is in `retaddr`. The `rcvmsg()` function is responsible for freeing `msg`. The response, if any, may be returned synchronously if `resp != NULL` by setting `*resp` to point to it. Generic control messages (except for `NGM_TEXT_STATUS`) are handled by the base system and need not be handled here.

Default action: Handle all generic control messages; otherwise returns `EINVAL`.

When to override: If you define any type-specific control messages, or you want to implement control messages defined by some other node type.

`int shutdown(node_p node);`

Purpose: Shutdown the node. Should disconnect all hooks by calling `ng_cutlinks()`, free all private per-node memory, release the assigned name (if any) via `ng_unname()`, and release the node itself by calling `ng_unref()` (this call releases the reference added by `ng_make_node_common()`).

In the case of persistent nodes, all hooks should be disconnected and the associated device (or whatever) reset, but the node should not be removed (i.e., only call `ng_cutlinks()`).

Default action: Calls `ng_cutlinks()`, `ng_unname()`, and `ng_unref()`.

When to override: When you need to undo the stuff you did in the constructor method.

`int newhook(node_p node, hook_p hook, const char *name);`

Purpose: Validate the connection of a hook and initialize any per-hook resources. The node should verify that the hook name is in fact one of the hook names supported by this node type. The uniqueness of the name will have already been verified (but it doesn't hurt to double-check).

If the hook requires per-hook information, this method should initialize `hook->private` accordingly.

Default action: Does nothing; the hook connection is always accepted.

When to override: Always, unless you plan to allow arbitrarily named hooks, have no per-hook initialization or resource allocation, and treat all hooks the same upon connection.

`hook_p findhook(node_p node, const char *name);`

Purpose: Find a connected hook on this node. It is not necessary to override this method unless the node supports a large number of hooks, where a linear search would be too slow.

Default action: Performs a linear search through the list of hooks connected to this node.

When to override: When your node supports a large number of simultaneously connected hooks (say, more than 50).

`int connect(hook_p hook);`

Purpose: Final verification of hook connection. This method gives the node a last chance to validate a newly connected hook. For example, the node may actually care who it's connected to. If this method returns an error, the connection is aborted.

Default action: Does nothing; the hook connection is accepted.

When to override: I've never had an occasion to override this method.

`int rcvdata(hook_p hook, struct mbuf *m, meta_p meta);`

Purpose: Receive an incoming data packet on a connected hook. The node is responsible for freeing the mbuf if it returns an error, or wishes to discard the data packet. Although not currently the case, in the future it could be that sometimes `m == NULL` (for example, if there is only a `meta` to be sent), so node types should handle this possibility.

Default action: Drops the data packet and meta-information.

When to override: Always, unless you intend to discard all received data packets.

`int rcvdataq(hook_p hook, struct mbuf *m, meta_p meta);`

Purpose: Queue an incoming data packet for reception on a connected hook. The node is responsible for freeing the mbuf if it returns an error, or wishes to discard the data packet.

The intention here is that some nodes may want to send data using a queuing mechanism instead of a functional mechanism. This requires cooperation of the receiving node type, which must implement this method in order for it to do anything different from `rcvdata()`.

Default action: Calls the `rcvdata()` method.

When to override: Never, unless you have a reason to treat incoming ``queue'' data differently from incoming ``non-queue'' data.

`int disconnect(hook_p hook);`

Purpose: Notification to the node that a hook is being disconnected. The node should release any per-hook resources allocated during `connect()`.

Although this function returns `int`, it should really return `void` because the return value is ignored; hook disconnection cannot be blocked by a node.

This function should check whether the last hook has been disconnected (`hook->node->numhooks == 0`) and if so, call `ng_rmnode()` to self-destruct, as is the custom. This helps avoid completely unconnected nodes that linger around in the system after their job is finished.

Default action: Does nothing.

When to override: Almost always.

`int mod_event(module_t mod, int what, void *arg);`

Purpose: Handle the events of loading and unloading the node type. Note that both events are handled through this one method, distinguished by `what` being either `MOD_LOAD` or `MOD_UNLOAD`. The `arg` parameter is a pointer to the `struct ng_type` defining the node type.

This method will never be called for `MOD_UNLOAD` when there are any nodes of this type currently in existence.

Currently, netgraph will only ever try to `MOD_UNLOAD` a node type when `kldunload(2)` is explicitly called. However, in the future more proactive unloading of node types may be implemented as a ``garbage collection'' measure.

Default action: Does nothing. If not overridden, `MOD_LOAD` and `MOD_UNLOAD` will succeed normally.

When to override: If your type needs to do any type-specific initialization or resource allocation upon loading, or undo any of that upon unloading. Also, if your type does not support unloading (perhaps because of unbreakable associations with other parts of the kernel) then returning an error in the `MOD_UNLOAD` case will prevent the type from being unloaded.

Netgraph header files

There are two header files all node types include. The netgraph.h header file defines the basic netgraph structures (good object-oriented design would dictate that the definitions of `struct ng_node` and `struct ng_hook` really don't belong here; instead, they should be private to the base netgraph code). Node structures are freed when the reference counter drops to zero after a call to `ng_unref()`. If a node has a name, that counts as a reference; to remove the name (and the reference), call `ng_unname()`. Of particular interest is `struct ng_type`, since every node type must supply one of these.

The ng_message.h header file defines structures and macros relevant to handling control messages. It defines the `struct ng_mesg` which every control message has as a prefix. It also serves as the ``public header file'' for all of the generic control messages, which all have typecookie `NGM_GENERIC_COOKIE`. The following summarizes the generic control messages:

`NGM_SHUTDOWN`	Disconnect all target node hooks and remove the node (or just reset if persistent)
`NGM_MKPEER`	Create a new node and connect to it
`NGM_CONNECT`	Connect a target node's hook to another node
`NGM_NAME`	Assign the target node a name
`NGM_RMHOOK`	Break a connection between the target node and another node
`NGM_NODEINFO`	Get information about the target node
`NGM_LISTHOOKS`	Get a list of all connected hooks on the target node
`NGM_LISTNAMES`	Get a list of all named nodes *
`NGM_LISTNODES`	Get a list of all nodes, named and unnamed *
`NGM_LISTTYPES`	Get a list of all installed node types *

`NGM_TEXT_STATUS`	Get a human readable status report from the target node (optional)
`NGM_BINARY2ASCII`	Convert a control message from binary to ASCII
`NGM_ASCII2BINARY`	Convert a control message from ASCII to binary

* Not node specific

For most of these commands, there are corresponding C structure(s) defined in `ng_message.h`.

The `netgraph.h` and `ng_message.h` header files also define several commonly used functions and macros:

`int ng_send_data(hook_p hook, struct mbuf *m, meta_p meta);`
: **What it does:** Delivers the mbuf `m` and associated meta-data `meta` out the hook `hook` and sets `error` to the resulting error code. Either or both of `m` and `meta` may be `NULL`. In all cases, the responsibility for freeing `m` and `meta` is lifted when this functions is called (even if there is an error), so these variables should be set to `NULL` after the call (this is done automatically if you use the `NG_SEND_DATA()` macro instead).

`int ng_send_dataq(hook_p hook, struct mbuf *m, meta_p meta);`
: **What it does:** Same as `ng_send_data()`, except the recipient node receives the data via its `rcvdataq()` method instead of its `rcvdata()` method. If the node type does not override `rcvdataq()`, then calling this is equivalent to calling `ng_send_data()`.

`int ng_queue_data(hook_p hook, struct mbuf *m, meta_p meta);`
: **What it does:** Same as `ng_send_data()`, except this is safe to call from a non-`splnet()` context. The mbuf and meta-information will be queued and delivered later at `splnet()`.

`int ng_send_msg(node_p here, struct ng_mesg *msg,`
`    const char *address, struct ng_mesg **resp);`
: **What it does:** Sends the netgraph control message pointed to by `msg` from the local node `here` to the node found at `address`, which may be an absolute or relative address. If `resp` is non-`NULL`, and the recipient node wishes to return a synchronous reply, it will set `*resp` to point at it. In this case, it is the calling node's responsibility to process and free `*resp`.

`int ng_queue_msg(node_p here, struct ng_mesg *msg, const char *address);`
: **What it does:** Same as `ng_send_msg()`, except this is safe to call from a non-`splnet()` context. The message will be queued and delivered later at `splnet()`. No synchronous reply is possible.

`NG_SEND_DATA(error, hook, m, meta)`
: **What it does:** Slightly safer version of `ng_send_data()`. This simply calls `ng_send_data()` and then sets `m` and `meta` to `NULL`. Either or both of `m` and `meta` may be `NULL`, though they must be actual variables (they can't be the constant `NULL` due to the way the macro works).

`NG_SEND_DATAQ(error, hook, m, meta)`
: **What it does:** Slightly safer version of `ng_send_dataq()`. This simply calls `ng_send_dataq()` and then sets `m` and `meta` to `NULL`. Either or both of `m` and `meta` may be `NULL`, though they must be actual variables (they can't be the constant `NULL` due to the way the macro works).

`NG_FREE_DATA(m, meta)`
: **What it does:** Frees `m` and `meta` and sets them to `NULL`. Either or both of `m` and `meta` may be `NULL`, though they must be actual variables (they can't be the constant `NULL` due to the way the macro works).

`NG_FREE_META(meta)`
: **What it does:** Frees `meta` and sets it to `NULL`. `meta` may be `NULL`, though it must be an actual variable (it can't be the constant `NULL` due to the way the macro works).

`NG_MKMESSAGE(msg, cookie, cmdid, len, how)`
: **What it does:** Allocates and initializes a new netgraph control message with `len` bytes of argument space (`len` should be zero if there are no arguments). `msg` should be of type `struct ng_mesg *`. The `cookie` and `cmdid` are the message typecookie and command ID. `how` is one of `M_WAIT` or `M_NOWAIT` (it's safer to use `M_NOWAIT`).

  Sets `msg` to NULL if memory allocation fails. Initializes the message token to zero.

`NG_MKRESPONSE(rsp, msg, len, how)`
: **What it does:** Allocates and initializes a new netgraph control message that is intended to be a response to `msg`. The response will have `len` bytes of argument space (`len` should be zero if there are no arguments). `msg` should be a pointer to an existing `struct ng_mesg` while `rsp` should be of type `struct ng_mesg *`. `how` is one of `M_WAIT` or `M_NOWAIT` (it's safer to use `M_NOWAIT`).

  Sets `rsp` to NULL if memory allocation fails.

`int ng_name_node(node_p node, const char *name);`
: **What it does:** Assign the global name `name` to node `node`. The name must be unique. This is often called from within node constructors for nodes that are associated with some other named kernel entity, e.g., a device or interface. Assigning a name to a node increments the node's reference count.

`void ng_cutlinks(node_p node);`
: **What it does:** Breaks all hook connections for `node`. Typically this is called during node shutdown.

`void ng_unref(node_p node);`
: **What it does:** Decrements a node's reference count, and frees the node if that count goes to zero. Typically this is called in the `shutdown()` method to release the reference created by `ng_make_node_common()`.

`void ng_unname(node_p node);`
: **What it does:** Removes the global name assigned to the node and

decrements the reference count. If the node does not have a name, this function has no effect. This should be called in the `shutdown()` method before freeing the node (via `ng_unref()`).

A real life example

Enough theory, let's see an example. Here is the implementation of the **tee** node type. As is the custom, the implementation consists of a public header file, a C file, and a man page. The header file is `ng_tee.h` and the C file is `ng_tee.c`.

Here are some things to notice about the header file:

- The header file defines the following important things:
  - The unique name of the type ``tee'' as `NG_TEE_NODE_TYPE`.
  - The unique typecookie for ``tee'' node specific control messages, `NGM_TEE_COOKIE`.
  - The names of the four hooks supported by ``tee'' nodes.
  - The two control messages understood by ``tee'' nodes, `NGM_TEE_GET_STATS` and `NGM_TEE_CLR_STATS`.
  - The structure returned by `NGM_TEE_GET_STATS`, which is a `struct ng_tee_stats`.

  This information is public because other node types need to know it in order to talk to and connect to tee nodes.

- Whenever there is an incompatible change in the control message format, the typecookie should be changed to avoid mysterious problems. The traditional way to generate unique typecookies is to use the output of ``date -u +%s''.
- Along with the C structures are corresponding macros that are used when converting between binary and ASCII. Although this information really belongs in the C file, it is put into the header file so it doesn't get out of sync with the actual structure.

Here are some things to notice about the C file:

- Nodes typically store information private to the node or to each hook. For the `ng_tee(8)` node type, this information is stored in a `struct privdata` for each node, and a `struct hookdata` for each hook.
- The `ng_tee_cmds` array defines how to convert the type specific control messages from binary to ASCII and back. See below.
- The `ng_tee_typestruct` at the beginning actually defines the node type for tee nodes. This structure contains the netgraph system version (to avoid incompatibilities), the unique type name (`NG_ECHO_NODE_TYPE`), pointers to the node type methods, and a pointer to the `ng_tee_cmds` array. Some methods don't need to be overridden because the default behavior is sufficient.
- The `NETGRAPH_INIT()` macro is required to link in the type. This macro works whether the node type is compiled as a KLD or directly into the kernel (in this case, using `options NETGRAPH_TEE`).
- Netgraph node structures (type `struct ng_node`) contain reference counts to ensure they get freed at the right time. A hidden side effect of calling `ng_make_node_common()` in the node constructor is that one reference is created. This reference is released by the `ng_unref()` call in the shutdown method `ngt_rmnode()`.
- Also in `ngt_rmnode()` is a call to `ng_bypass()`. This is a bit of a kludge that joins two edges by disconnecting the node in between them (in this case, the tee node).
- Note that in the function `ngt_disconnect()` the node destroys itself when the last hook is disconnected. This keeps nodes from lingering around after they have nothing left to do.
- No spl synchronization calls are necessary; the entire thing runs at `splnet()`.

Converting control messages to/from ASCII

Netgraph provides an easy way to convert control messages (indeed, any C structure) between binary and ASCII formats. A detailed explanation is beyond the scope of this article, but here we'll give an overview.

Recall that control messages have a fixed header (`struct ng_mesg`) followed by a variable length payload having arbitrary structure and contents. In addition, the control message header contains a flag bit indicating whether the messages is a command or a reply. Usually the payload will be structured differently in the command and the response. For example, the ``tee'' node has a `NGM_TEE_GET_STATS` control message. When sent as a command (`(msg->header.flags & NGF_RESP) == 0`), the payload is empty. When sent as a response to a command (`(msg->header.flags & NGF_RESP) != 0`), the payload contains a `struct ng_tee_stats` that contains the node statistics.

So for each control message that a node type understands, the node type defines how to convert the payload area of that control message (in both cases, command and response) between its native binary representation and a human-readable ASCII version. These definitions are called **netgraph parse types**.

The `cmdlist` field in the `struct ng_type` that defines a node type is a pointer to an array of `struct ng_cmdlists`. Each element in this array corresponds to a type-specific control message understood by this node. Along with the typecookie and command ID (which uniquely identify the control message), are an ASCII name and two netgraph parse types that define how the payload area data is structured i.e. one for each direction (command and response).

Parse types are built up from the predefined parse types defined in `ng_parse.h`. Using these parse types, you can describe any arbitrarily complicated C structure, even one containing variable length arrays and strings. The ``tee'' node type has an example of doing this for the `struct ng_tee_stats` returned by the `NGM_TEE_GET_STATS` control message (see `ng_tee.h` and `ng_tee.c`).

You can also define your own parse types from scratch if necessary. For example, the ``ksocket'' node type contains special code for converting a `struct sockaddr` in the address families `AF_INET` and `AF_LOCAL`, to make them more human friendly. The relevant code can be found in `ng_ksocket.h` and `ng_ksocket.c`, specifically the section labeled ``STRUCT SOCKADDR PARSE TYPE''.

Parse types are a convenient and efficient way to effect binary/ASCII conversion in the kernel without a lot of manual parsing code and string manipulation. When performance is a real issue, binary control messages can always be used directly to avoid any conversion.

The gory details about parse types are available in `ng_parse.h` and `ng_parse.c`.

Programming gotcha's

Some things to look out for if you plan on implementing your own netgraph node type:

- First, make sure you fully understand how **mbuf**'s work and are used.
- All data packets must be **packet header mbuf**'s, i.e., with the `M_PKTHDR` flag set.
- Be careful to always update `m->m_pkthdr.len` when you update `m->m_len` for any mbuf in the chain.
- Be careful to check `m->m_len` and call `m_pullup()` if necessary before accessing mbuf data. **Don't** call `m_pullup()` unless necessary. You should always follow this pattern:

```
struct foobar *f;

if (m->m_len < sizeof(*f) && (m = m_pullup(m, sizeof(*f))) == NULL)
    NG_FREE_META(meta);
    return (ENOBUFS);
}
f=mtod(m, struct foobar *);
...
```

- Be careful to release all resources at the appropriate time, e.g., during the `disconnect()` and `shutdown()` methods to avoid memory leaks, etc. I've accidentally done things like leave timers running with disastrous results.
- If you use a timer (see `timeout(9)`), be sure to set `splnet()` as the first thing in your handler (and `splx()` before exiting, of course). The `timeout()` routine does not preserve the SPL level to the event handler.
- Make sure your node disappears when all edges have been broken unless there's a good reason not to.

Part IV: Future Directions

Netgraph is still a work in progress, and contributors are welcome! Here are some ideas for future work.

Node types

There are many node types yet to be written:

- A ``slip'' node type that implements the SLIP protocol. This should be pretty easy and may be done soon.
- More PPP compression and encryption nodes that can connect to a `ng_ppp(8)` node, e.g., PPP Deflate compression, PPP 3DES encryption, etc.
- An implementation of `ipfw(4)` as a netgraph node.
- An implementation of the Dynamic Packet Filter as a netgraph node. DPF is sort of a hyper-speed JIT compiling version of BPF.
- A generic ``mux'' node type, where each hook could be configured with a unique header to append/strip from data packets.

FreeBSD currently has **four** PPP implementations: `sppp(4)`, `pppd(8)`, `ppp(8)`, and the MPD port. This is pretty silly. Using netgraph, these can all be collapsed into a single user-land daemon that handles all the configuration and negotiation, while routing all data strictly in the kernel via `ng_ppp(8)` nodes. This combines the flexibility and configuration benefits of the user-land daemons with the speed of the kernel implementations. Right now MPD is the only implementation that has been fully ``netgraphified'' but plans are in the works for `ppp(8)` as well.

Control message ASCII-fication

Not all node types that define their own control messages support conversion between binary and ASCII. One project is to finish this work for those nodes that still need it.

Control flow

One issue that may need addressing is control flow. Right now when you send a data packet, if the ultimate recipient of that node can't handle it because of a full transmit queue or something, all it can do is drop the packet and return `ENOBUFS`. Perhaps we can define a new return code `ESLOWDOWN` or something that means ``data packet not dropped; queue full; slow down and try again later.'' Another possibility would be to define meta-data types for the equivalents of XOFF (stop flow) and XON (restart flow).

Code cleanups

Netgraph is somewhat object oriented, but could benefit from a more rigorous object oriented design without suffering too much in performance. There are still too many visible structure fields that shouldn't be accessible, etc., as well as other miscellaneous code cleanups.

Also, all of the node type man pages (e.g., `ng_tee(8)`) really belong in section 4 rather than section 8.

Electrocution

It would be nice to have a new generic control message `NGM_ELECTROCUTE`, which when sent to a node would shutdown that node as well as every node it was connected to, and every node those nodes were connected to, etc. This would allow for a quick cleanup of an arbitrarily complicated netgraph graph in a single blow. In addition, there might be a new socket option (see `setsockopt(2)`) that you could set on a `ng_socket(8)` socket that would cause an `NGM_ELECTROCUTE` to be automatically generated when the socket was closed.

Together, these two features would lead to more reliable avoidance of netgraph ``node leak.''

Infinite loop detection

It would be easy to include ``infinite loop detection'' in the base netgraph code. That is, each node would have a private counter. The counter would be incremented before each call to a node's `rcvdata()` method, and decremented afterwards. If the counter reached some insanely high value, then we've detected an infinite loop (and avoided a kernel panic).

New node types

There are lots of new and improved node types that could be created, for example:

- A routing node type. Each connected hook would correspond to a route destination, i.e., an address and netmask combination. The routes would be managed via control messages.
- A stateful packet filtering/firewall/address translation node type (replacement for ipfw and/or ipfirewall)
- Node type for bandwidth limiting and/or bandwidth accounting
- Adding VLAN support to the existing Ethernet nodes.

If you really wanted to get crazy

In theory, the BSD networking subsystem could be entirely replaced by netgraph. Of course, this will probably never happen, but it makes for a nice thought experiment. Each networking device would be a persistent netgraph node (like Ethernet devices are now). On top of each Ethernet device node would be an ``Ethertype multiplexor.'' Connected to this would be IP, ARP, IPX, AppleTalk, etc. nodes. The IP node would be a simple ``IP protocol multiplexor'' node on top of which would sit TCP, UDP, etc. nodes. The TCP and UDP nodes would in turn have socket-like nodes on top of them. Etc, etc.

Other crazy ideas (disclaimer: these are crazy ideas):

- Make *all* devices appear as netgraph nodes. Convert between `ioctl(2)`'s and control messages. Talk directly to your SCSI disk with `ngctl(8)`! Seamless integration between netgraph and DEVFS.
- A netgraph node that is also a VFS layer? A filesystem view of the space of netgraph nodes?
- If NFS can work over UDP, it can work over netgraph. You could have NFS disks remotely mounted via an ATM link, or simply do NFS over raw Ethernet and cut out the UDP middleman.
- A ``programmable'' node type whose implementation would depend on its configuration using some kind of node pseudo-code.

Surely there are lots more crazy ideas we haven't thought of yet.

Manual Pages

ngctl(8)
nghook(8)
netgraph(4)

http://www.freebsd.org/cgi/man.cgi

NAME
ngctl -- netgraph control utility

SYNOPSIS
ngctl [**-d**] [**-f** *filename*] [**-n** *nodename*] [*command ...*]

DESCRIPTION
The **ngctl** utility creates a new netgraph node of type *socket* which can be used to issue netgraph commands. If no **-f** flag is given, no command is supplied on the command line, and standard input is a tty, **ngctl** will enter interactive mode. Otherwise **ngctl** will execute the supplied command(s) and exit immediately.

Nodes can be created, removed, joined together, etc. ASCII formatted control messages can be sent to any node if that node supports binary/ASCII control message conversion.

In interactive mode, **ngctl** will display any control messages and data packets received by the socket node. In the case of control messages, the message arguments are displayed in ASCII form if the originating node supports conversion.

The options are as follows:

-f *nodeinfo*

Read commands from the named file. A single dash represents the standard input. Blank lines and lines starting with a ``#'' are ignored.

-n *nodename*
Assign *nodename* to the newly created netgraph node. The default name is *ngctlXXX* where XXX is the process ID number.

-d Increase the debugging verbosity level.

COMMANDS
The currently supported commands in **ngctl** are:

config	get or set configuration of node at <path>
connect	Connects hook <peerhook> of the node at <relpath> to <hook>
debug	Get/set debugging verbosity level
dot	Produce a GraphViz (.dot) of the entire netgraph.
help	Show command summary or get more help on a specific command
list	Show information about all nodes
mkpeer	Create and connect a new node to the node at "path"
msg	Send a netgraph control message to the node at "path"
name	Assign name <name> to the node at <path>
read	Read and execute commands from a file
rmhook	Disconnect hook "hook" of the node at "path"
show	Show information about the node at <path>
shutdown	Shutdown the node at <path>
status	Get human readable status information from the node at <path>
types	Show information about all installed node types
write	Send a data packet down the hook named by "hook".
quit	Exit program

Some commands have aliases, e.g., ``ls'' is the same as ``list''. The ``help'' command displays the available commands, their usage and aliases, and a brief description.

EXIT STATUS
The **ngctl** utility exits 0 on success, and >0 if an error occurs.

SEE ALSO
netgraph(3), netgraph(4), nghook(8)

HISTORY
The **netgraph** system was designed and first implemented at Whistle Communications, Inc. in a version of FreeBSD 2.2 customized for the Whistle InterJet.

AUTHORS
Archie Cobbs <archie@whistle.com>

NAME
nghook -- connect to a netgraph(4) node

SYNOPSIS
nghook [**-adlnSs**] [**-m** *msg*] *path* [*hookname*]
nghook **-e** [**-n**] [**-m** *msg*] *path hookname program* [*args ...*]

DESCRIPTION
The **nghook** utility creates a ng_socket(4) socket type node and connects it to hook *hookname* of the node found at *path*. If *hookname* is omitted, ``debug'' is assumed.

If the **-e** option is given, the third argument is interpreted as the path to a program, and this program is executed with the remaining arguments as its arguments. Before executing, the program Netgraph messages (specified by the **-m** option) are sent to the node. The program is executed with its standard input (unless closed by **-n**) and output connected to the hook.

If the **-e** option is not given, all data written to standard input is sent to the node, and all data received from the node is relayed to standard output. Messages specified with **-m** are sent to the node before the loop is entered. The **nghook** utility exits when EOF is detected on standard input in this case.

The options are as follows:

-a Output each packet read in human-readable decoded ASCII form instead of raw binary.

-d Increase the debugging verbosity level.

-e Execute the program specified by the third argument.

-l Loops all received data back to the hook in addition to writing it to standard output.

-m *msg* Before executing the program (in **-e** mode) send the given ASCII control message to the node. This option may be given more than once.

-n Do not attempt to read any data from standard input. The **nghook**

utility will continue reading from the node until stopped by a signal.

-S Use file descriptor 0 for output instead of the default 1.

-s Use file descriptor 1 for input instead of the default 0.

SEE ALSO
netgraph(3), netgraph(4), ngctl(8)

HISTORY
The **netgraph** system was designed and first implemented at Whistle Communications, Inc. in a version of FreeBSD 2.2 customized for the Whistle InterJet.

AUTHORS
Archie Cobbs <archie@whistle.com>

BUGS

Although all input is read in unbuffered mode, there is no way to control the packetization of the input.

If the node sends a response to a message (specified by **-m**), this response is lost.

NAME

netgraph -- graph based kernel networking subsystem

DESCRIPTION

The **netgraph** system provides a uniform and modular system for the implementation of kernel objects which perform various networking functions. The objects, known as *nodes*, can be arranged into arbitrarily complicated graphs. Nodes have *hooks* which are used to connect two nodes together, forming the edges in the graph. Nodes communicate along the edges to process data, implement protocols, etc.

The aim of **netgraph** is to supplement rather than replace the existing kernel networking infrastructure. It provides:

- A flexible way of combining protocol and link level drivers.
- A modular way to implement new protocols.
- A common framework for kernel entities to inter-communicate.
- A reasonably fast, kernel-based implementation.

Nodes and Types

The most fundamental concept in **netgraph** is that of a *node*. All nodes implement a number of predefined methods which allow them to interact with other nodes in a well defined manner.

Each node has a *type*, which is a static property of the node determined at node creation time. A node's type is described by a unique ASCII type name. The type implies what the node does and how it may be connected to other nodes.

In object-oriented language, types are classes, and nodes are instances of their respective class. All node types are subclasses of the generic node type, and hence inherit certain common functionality and capabilities (e.g., the ability to have an ASCII name).

Nodes may be assigned a globally unique ASCII name which can be used to refer to the node. The name must not contain the characters \`.' or \`:', and is limited to NG_NODESIZ characters (including the terminating NUL character).

Each node instance has a unique *ID number* which is expressed as a 32-bit hexadecimal value. This value may be used to refer to a node when there is no ASCII name assigned to it.

Hooks

Nodes are connected to other nodes by connecting a pair of *hooks*, one from each node. Data flows bidirectionally between nodes along connected pairs of hooks. A node may have as many hooks as it needs, and may assign whatever meaning it wants to a hook.

Hooks have these properties:

- A hook has an ASCII name which is unique among all hooks on that node (other hooks on other nodes may have the same name). The name must not contain the characters \`.' or \`:', and is limited to NG_HOOKSIZ characters (including the terminating NUL character).

- A hook is always connected to another hook. That is, hooks are created at the time they are connected, and breaking an edge by removing either hook destroys both hooks.

- A hook can be set into a state where incoming packets are always queued by the input queueing system, rather than being delivered directly. This can be used when the data is sent from an interrupt handler, and processing must be quick so as not to block other interrupts.

- A hook may supply overriding receive data and receive message functions, which should be used for data and messages received through that hook in preference to the general node-wide methods.

A node may decide to assign special meaning to some hooks. For example, connecting to the hook named *debug* might trigger the node to start sending debugging information to that hook.

Data Flow

Two types of information flow between nodes: data messages and control messages. Data messages are passed in *mbuf chains* along the edges in the graph, one edge at a time. The first *mbuf* in a chain must have the M_PKTHDR flag set. Each node decides how to handle data received through one of its hooks.

Along with data, nodes can also receive control messages. There are generic and type-specific control messages. Control messages have a common header format, followed by type-specific data, and are binary structures for efficiency. However, node types may also support conversion of the type-specific data between binary and ASCII formats, for debugging and human interface purposes (see the NGM_ASCII2BINARY and NGM_BINARY2ASCII generic control messages below). Nodes are not required to support these conversions.

There are three ways to address a control message. If there is a sequence of edges connecting the two nodes, the message may be \`\`source routed'' by specifying the corresponding sequence of ASCII hook names as the destination address for the message (relative addressing). If the destination is adjacent to the source, then the source node may simply specify (as a pointer in the code) the hook across which the message should be sent. Otherwise, the recipient node's global ASCII name (or equivalent ID-based name) is used as the destination address for the message (absolute addressing). The two types of ASCII addressing may be combined, by specifying an absolute start node and a sequence of hooks. Only the ASCII addressing modes are available to control programs outside the kernel; use of direct pointers is limited to kernel modules.

Messages often represent commands that are followed by a reply message in the reverse direction. To facilitate this, the recipient of a control message is supplied with a \`\`return address'' that is suitable for addressing a reply.

Each control message contains a 32-bit value, called a \`\`typecookie'', indicating the type of the message, i.e. how to interpret it. Typically each type defines a unique typecookie for the messages that it understands. However, a node may choose to recognize and implement more than one type of messages.

If a message is delivered to an address that implies that it arrived at that node through a particular hook (as opposed to having been directly addressed using its ID or global name) then that hook is identified to the receiving node. This allows a message to be re-routed or passed on, should a node decide that this is required, in much the same way that data packets are passed around between nodes. A set of standard messages for flow control and link management purposes are defined by the base system that are usually passed around in this manner. Flow control message would usually travel in the opposite direction to the data to which they pertain.

Netgraph is (Usually) Functional

In order to minimize latency, most **netgraph** operations are functional. That is, data and control messages are delivered by making function calls rather than by using queues and mailboxes. For example, if node A wishes to send a data *mbuf* to neighboring node B, it calls the generic **netgraph** data delivery function. This function in turn locates node B and calls B's ``receive data'' method. There are exceptions to this.

Each node has an input queue, and some operations can be considered to be *writers* in that they alter the state of the node. Obviously, in an SMP world it would be bad if the state of a node were changed while another data packet were transiting the node. For this purpose, the input queue implements a *reader/writer* semantic so that when there is a writer in the node, all other requests are queued, and while there are readers, a writer, and any following packets are queued. In the case where there is no reason to queue the data, the input method is called directly, as mentioned above.

A node may declare that all requests should be considered as writers, or that requests coming in over a particular hook should be considered to be a writer, or even that packets leaving or entering across a particular hook should always be queued, rather than delivered directly (often useful for interrupt routines who want to get back to the hardware quickly). By default, all control message packets are considered to be writers unless specifically declared to be a reader in their definition. (See NGM_READONLY in <*ng_message.h*>.)

While this mode of operation results in good performance, it has a few implications for node developers:

- Whenever a node delivers a data or control message, the node may need to allow for the possibility of receiving a returning message before the original delivery function call returns.

- **Netgraph** provides internal synchronization between nodes. Data always enters a ``graph'' at an *edge node*. An *edge node* is a node that interfaces between **netgraph** and some other part of the system. Examples of ``edge nodes'' include device drivers, the *socket*, *ether*, *tty*, and *ksocket* node type. In these *edge nodes*, the calling thread directly executes code in the node, and from that code calls upon the **netgraph** framework to deliver data across some edge in the graph. From an execution point of view, the calling thread will execute the **netgraph** framework methods, and if it can acquire a lock to do so, the input methods of the next node. This continues until either the data is discarded or queued for some device or system entity, or the thread is unable to acquire a lock on the next node. In that case, the data is queued for the node, and execution rewinds back to the original calling entity. The queued data will be picked up and processed by either the current holder of the lock when they have completed their operations, or by a special **netgraph** thread that is activated when there are such items queued.

- It is possible for an infinite loop to occur if the graph contains cycles.

So far, these issues have not proven problematical in practice.

Interaction with Other Parts of the Kernel

A node may have a hidden interaction with other components of the kernel outside of the **netgraph** subsystem, such as device hardware, kernel protocol stacks, etc. In fact, one of the benefits of **netgraph** is the ability to join disparate kernel networking entities together in a consistent communication framework.

An example is the *socket* node type which is both a **netgraph** node and a socket(2) in the protocol family PF_NETGRAPH. Socket nodes allow user processes to participate in **netgraph**. Other nodes communicate with socket nodes using the usual methods, and the node hides the fact that it is also passing information to and from a cooperating user process.

Another example is a device driver that presents a node interface to the hardware.

Node Methods

Nodes are notified of the following actions via function calls to the following node methods, and may accept or reject that action (by returning the appropriate error code):

Creation of a new node
: The constructor for the type is called. If creation of a new node is allowed, constructor method may allocate any special resources it needs. For nodes that correspond to hardware, this is typically done during the device attach routine. Often a global ASCII name corresponding to the device name is assigned here as well.

Creation of a new hook
: The hook is created and tentatively linked to the node, and the node is told about the name that will be used to describe this hook. The node sets up any special data structures it needs, or may reject the connection, based on the name of the hook.

Successful connection of two hooks
: After both ends have accepted their hooks, and the links have been made, the nodes get a chance to find out who their peer is across the link, and can then decide to reject the connection. Tear-down is automatic. This is also the time at which a node may decide whether to set a particular hook (or its peer) into the *queueing* mode.

Destruction of a hook
: The node is notified of a broken connection. The node may consider some hooks to be critical to operation and others to be expendable: the disconnection of one hook may be an acceptable event while for another it may effect a total shutdown for the node.

Preshutdown of a node
: This method is called before real shutdown, which is discussed below. While in this method, the node is fully operational and can send a ``goodbye'' message to its peers, or it can exclude itself from the chain and reconnect its peers together, like the ng_tee(4) node type does.

Shutdown of a node
: This method allows a node to clean up and to ensure that any actions that need to be performed at this time are taken. The method is called by the generic (i.e., superclass) node destructor which will get rid of the generic components of the node. Some nodes (usually associated with a piece of hardware) may be *persistent* in that a shutdown breaks all edges and resets the node, but does not remove it. In this case, the shutdown method should not free its resources, but rather, clean up and then call the **NG_NODE_REVIVE**() macro to signal the generic code that the shutdown is aborted. In the case where the shutdown is started by the node itself due to hardware removal or unloading (via **ng_rmnode_self**()), it should set the NGF_REALLY_DIE flag to signal to its own shutdown method that it is not to persist.

Sending and Receiving Data

Two other methods are also supported by all nodes:

Receive data message
: A **netgraph** *queueable request item*, usually referred to as an *item*, is received by this function. The item contains a pointer to an *mbuf*.

: The node is notified on which hook the item has arrived, and can use this information in its processing decision. The receiving node must always **NG_FREE_M**() the *mbuf chain* on completion or error, or pass it on to another node (or kernel module) which will then be responsible for freeing it. Similarly, the *item* must be freed if it is not to be passed on to another node, by using the **NG_FREE_ITEM**() macro. If the item still holds references to *mbufs* at the time of freeing then they will also be appropriately freed. Therefore, if there is any chance that the *mbuf* will be changed or freed separately from the item, it is very important that it be retrieved using the **NGI_GET_M**() macro that also removes the reference within the item. (Or multiple frees of the same object will occur.)

If it is only required to examine the contents of the *mbufs*, then it is possible to use the **NGI_M**() macro to both read and rewrite *mbuf*

pointer inside the item.

If developer needs to pass any meta information along with the *mbuf chain*, he should use mbuf_tags(9) framework. **Note that old netgraph**

specific meta-data format is obsoleted now.

The receiving node may decide to defer the data by queueing it in the **netgraph** NETISR system (see below). It achieves this by setting the HK_QUEUE flag in the flags word of the hook on which that data will arrive. The infrastructure will respect that bit and queue the data for delivery at a later time, rather than deliver it directly. A node may decide to set the bit on the *peer* node, so that its own output packets are queued.

The node may elect to nominate a different receive data function for data received on a particular hook, to simplify coding. It uses the **NG_HOOK_SET_RCVDATA**(*hook*, *fn*) macro to do this. The function receives the same arguments in every way other than it will receive all (and only) packets from that hook.

Receive control message

This method is called when a control message is addressed to the node. As with the received data, an *item* is received, with a pointer to the control message. The message can be examined using the **NGI_MSG**() macro, or completely extracted from the item using the **NGI_GET_MSG**() which also removes the reference within the item. If the item still holds a reference to the message when it is freed (using the **NG_FREE_ITEM**() macro), then the message will also be freed appropriately. If the reference has been removed, the node must free the message itself using the **NG_FREE_MSG**() macro. A return address is always supplied, giving the address of the node that originated the message so a reply message can be sent anytime later. The return address is retrieved from the *item* using the **NGI_RETADDR**() macro and is of type *ng_ID_t*. All control messages and replies are allocated with the malloc(9) type M_NETGRAPH_MSG, however it is more convenient to use the **NG_MKMESSAGE**() and **NG_MKRESPONSE**() macros to allocate and fill out a message. Messages must be freed using the **NG_FREE_MSG**() macro.

If the message was delivered via a specific hook, that hook will also be made known, which allows the use of such things as flow-control messages, and status change messages, where the node may want to forward the message out another hook to that on which it arrived.

The node may elect to nominate a different receive message function for messages received on a particular hook, to simplify coding. It uses the **NG_HOOK_SET_RCVMSG**(*hook*, *fn*) macro to do this. The function receives the same arguments in every way other than it will receive all (and only) messages from that hook.

Much use has been made of reference counts, so that nodes being freed of all references are automatically freed, and this behaviour has been tested and debugged to present a consistent and trustworthy framework for the ``type module'' writer to use.

Addressing

The **netgraph** framework provides an unambiguous and simple to use method of specifically addressing any single node in the graph. The naming of a node is independent of its type, in that another node, or external component need not know anything about the node's type in order to address it so as to send it a generic message type. Node and hook names should be chosen so as to make addresses meaningful.

Addresses are either absolute or relative. An absolute address begins with a node name or ID, followed by a colon, followed by a sequence of hook names separated by periods. This addresses the node reached by starting at the named node and following the specified sequence of hooks. A relative address includes only the sequence of hook names, implicitly starting hook traversal at the local node.

There are a couple of special possibilities for the node name. The name `.' (referred to as `.:') always refers to the local node. Also, nodes that have no global name may be addressed by their ID numbers, by enclosing the hexadecimal representation of the ID number within the square brackets. Here are some examples of valid **netgraph** addresses:

```
.:
[3f]:
foo:
.:hook1
foo:hook1.hook2
[d80]:hook1
```

The following set of nodes might be created for a site with a single physical frame relay line having two active logical DLCI channels, with RFC 1490 frames on DLCI 16 and PPP frames over DLCI 20:

```
[type SYNC ]                  [type FRAME]                 [type RFC1490]
[ "Frame1" ](uplink)<-->(data)[<un-named>](dlci16)<-->(mux)[<un-named>  ]
[    A     ]                  [    B     ](dlci20)<---+    [     C      ]
                                                      |
                                                      |      [ type PPP ]
                                                      +>(mux)[<un-named>]
                                                             [    D     ]
```

One could always send a control message to node C from anywhere by using the name ``Frame1:uplink.dlci16''. In this case, node C would also be notified that the message reached it via its hook *mux*. Similarly, ``Frame1:uplink.dlci20'' could reliably be used to reach node D, and node A could refer to node B as ``.:uplink'', or simply ``uplink''. Conversely, B can refer to A as ``data''. The address ``mux.data'' could be used by both nodes C and D to address a message to node A.

Note that this is only for *control messages*. In each of these cases, where a relative addressing mode is used, the recipient is notified of the hook on which the message arrived, as well as the originating node. This allows the option of hop-by-hop distribution of messages and state information. Data messages are *only* routed one hop at a time, by specifying the departing hook, with each node making the next routing decision. So when B receives a frame on hook *data*, it decodes the frame relay header to determine the DLCI, and then forwards the unwrapped frame to either C or D.

In a similar way, flow control messages may be routed in the reverse direction to outgoing data. For example a ``buffer nearly full'' message from ``Frame1:'' would be passed to node B which might decide to send similar messages to both nodes C and D. The nodes would use *direct hook* *pointer* addressing to route the messages. The message may have travelled from ``Frame1:'' to B as a synchronous reply, saving time and cycles.

Netgraph Structures

Structures are defined in <*netgraph/netgraph.h*> (for kernel structures only of interest to nodes) and <*netgraph/ng_message.h*> (for message definitions also of interest to user programs).

The two basic object types that are of interest to node authors are *nodes*

and *hooks*. These two objects have the following properties that are also of interest to the node writers.

struct ng_node
: Node authors should always use the following **typedef** to declare their pointers, and should never actually declare the structure.

typedef struct ng_node *node_p;

The following properties are associated with a node, and can be accessed in the following manner:

Validity
: A driver or interrupt routine may want to check whether the node is still valid. It is assumed that the caller holds a reference on the node so it will not have been freed, however it may have been disabled or otherwise shut down. Using the **NG_NODE_IS_VALID**(*node*) macro will return this state. Eventually it should be almost impossible for code to run in an invalid node but at this time that work has not been completed.

Node ID (*ng_ID_t*)
: This property can be retrieved using the macro **NG_NODE_ID**(*node*).

Node name
: Optional globally unique name, NUL terminated string. If there is a value in here, it is the name of the node.

```
if (NG_NODE_NAME(node)[0] != '\0') ...

if (strcmp(NG_NODE_NAME(node), "fred") == 0) ...
```

A node dependent opaque cookie
: Anything of the pointer type can be placed here. The macros **NG_NODE_SET_PRIVATE**(*node*, *value*) and **NG_NODE_PRIVATE**(*node*) set and retrieve this property, respectively.

Number of hooks
: The **NG_NODE_NUMHOOKS**(*node*) macro is used to retrieve this value.

Hooks
: The node may have a number of hooks. A traversal method is provided to allow all the hooks to be tested for some condition. **NG_NODE_FOREACH_HOOK**(*node*, *fn*, *arg*, *rethook*) where *fn* is a function that will be called for each hook with the form **fn**(*hook*, *arg*) and returning 0 to terminate the search. If the search is terminated, then *rethook* will be set to the hook at which the search was terminated.

struct ng_hook

Node authors should always use the following **typedef** to declare their hook pointers.

typedef struct ng_hook *hook_p;

The following properties are associated with a hook, and can be accessed in the following manner:

A hook dependent opaque cookie
: Anything of the pointer type can be placed here. The macros **NG_HOOK_SET_PRIVATE**(*hook*, *value*) and **NG_HOOK_PRIVATE**(*hook*) set and retrieve this property, respectively.

An associate node
: The macro **NG_HOOK_NODE**(*hook*) finds the associated node.

A peer hook (*hook_p*)
: The other hook in this connected pair. The **NG_HOOK_PEER**(*hook*) macro finds the peer.

References
: The **NG_HOOK_REF**(*hook*) and **NG_HOOK_UNREF**(*hook*) macros increment and decrement the hook reference count accordingly. After decrement you should always assume the hook has been freed unless you have another reference still valid.

Override receive functions
: The **NG_HOOK_SET_RCVDATA**(*hook*, *fn*) and **NG_HOOK_SET_RCVMSG**(*hook*, *fn*) macros can be used to set override methods that will be used in preference to the generic receive data and receive message functions. To unset these, use the macros to set them to NULL. They will only be used for data and messages received on the hook on which they are set.

The maintenance of the names, reference counts, and linked list of hooks for each node is handled automatically by the **netgraph** subsystem. Typically a node's private info contains a back-pointer to the node or hook structure, which counts as a new reference that must be included in the reference count for the node. When the node constructor is called, there is already a reference for this calculated in, so that when the node is destroyed, it should remember to do a **NG_NODE_UNREF**() on the node.

From a hook you can obtain the corresponding node, and from a node, it is possible to traverse all the active hooks.

A current example of how to define a node can always be seen in *src/sys/netgraph/ng_sample.c* and should be used as a starting point for new node writers.

Netgraph Message Structure

Control messages have the following structure:

```
#define NG_CMDSTRSIZ    32      /* Max command string (including nul) */

struct ng_mesg {
  struct ng_msghdr {
    u_char      version;        /* Must equal NG_VERSION */
    u_char      spare;          /* Pad to 2 bytes */
    u_short     arglen;         /* Length of cmd/resp data */
    u_long      flags;          /* Message status flags */
    u_long      token;          /* Reply should have the same token */
    u_long      typecookie;     /* Node type understanding this message */
    u_long      cmd;            /* Command identifier */
    u_char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
  } header;
  char  data[0];                /* Start of cmd/resp data */
};

#define NG_ABI_VERSION  5               /* Netgraph kernel ABI version */
#define NG_VERSION      4               /* Netgraph message version */
#define NGF_ORIG        0x0000          /* Command */
```

```
#define NGF_RESP        0x0001          /* Response */
```

Control messages have the fixed header shown above, followed by a variable length data section which depends on the type cookie and the command. Each field is explained below:

version
: Indicates the version of the **netgraph** message protocol itself. The current version is NG_VERSION.

arglen
: This is the length of any extra arguments, which begin at *data*.

flags
: Indicates whether this is a command or a response control message.

token
: The *token* is a means by which a sender can match a reply message to the corresponding command message; the reply always has the same token.

typecookie
: The corresponding node type's unique 32-bit value. If a node does not recognize the type cookie it must reject the message by returning EINVAL.

  Each type should have an include file that defines the commands, argument format, and cookie for its own messages. The typecookie ensures that the same header file was included by both sender and receiver; when an incompatible change in the header file is made, the typecookie *must* be changed. The de-facto method for generating unique type cookies is to take the seconds from the Epoch at the time the header file is written (i.e., the output of ``**date -u** +%s'').

  There is a predefined typecookie NGM_GENERIC_COOKIE for the *generic* node type, and a corresponding set of generic messages which all nodes understand. The handling of these messages is automatic.

cmd
: The identifier for the message command. This is type specific, and is defined in the same header file as the typecookie.

cmdstr
: Room for a short human readable version of *command* (for debugging purposes only).

Some modules may choose to implement messages from more than one of the header files and thus recognize more than one type cookie.

Control Message ASCII Form

Control messages are in binary format for efficiency. However, for debugging and human interface purposes, and if the node type supports it, control messages may be converted to and from an equivalent ASCII form. The ASCII form is similar to the binary form, with two exceptions:

1. The *cmdstr* header field must contain the ASCII name of the command, corresponding to the *cmd* header field.

2. The arguments field contains a NUL-terminated ASCII string version of the message arguments.

In general, the arguments field of a control message can be any arbitrary C data type. **Netgraph** includes parsing routines to support some predefined datatypes in ASCII with this simple syntax:

- Integer types are represented by base 8, 10, or 16 numbers.
- Strings are enclosed in double quotes and respect the normal C language backslash escapes.
- IP addresses have the obvious form.
- Arrays are enclosed in square brackets, with the elements listed consecutively starting at index zero. An element may have an optional index and equals sign (`=') preceding it. Whenever an element does not have an explicit index, the index is implicitly the previous element's index plus one.
- Structures are enclosed in curly braces, and each field is specified in the form *fieldname*=*value*.
- Any array element or structure field whose value is equal to its ``default value'' may be omitted. For integer types, the default value is usually zero; for string types, the empty string.
- Array elements and structure fields may be specified in any order.

Each node type may define its own arbitrary types by providing the necessary routines to parse and unparse. ASCII forms defined for a specific node type are documented in the corresponding man page.

Generic Control Messages

There are a number of standard predefined messages that will work for any node, as they are supported directly by the framework itself. These are defined in <*netgraph/ng_message.h*> along with the basic layout of messages and other similar information.

NGM_CONNECT
: Connect to another node, using the supplied hook names on either end.

NGM_MKPEER
: Construct a node of the given type and then connect to it using the supplied hook names.

NGM_SHUTDOWN
: The target node should disconnect from all its neighbours and shut down. Persistent nodes such as those representing physical hardware might not disappear from the node namespace, but only reset themselves. The node must disconnect all of its hooks. This may result in neighbors shutting themselves down, and possibly a cascading shutdown of the entire connected graph.

NGM_NAME
: Assign a name to a node. Nodes can exist without having a name, and this is the default for nodes created using the NGM_MKPEER method. Such nodes can only be addressed relatively or by their ID number.

NGM_RMHOOK
: Ask the node to break a hook connection to one of its neighbours. Both nodes will have their ``disconnect'' method invoked. Either node may elect to totally shut down as a result.

NGM_NODEINFO
: Asks the target node to describe itself. The four returned fields are the node name (if named), the node type, the node ID and the number of hooks attached. The ID is an internal number unique to that node.

NGM_LISTHOOKS
: This returns the information given by NGM_NODEINFO, but in addition includes an array of fields describing each link, and the

description for the node at the far end of that link.

NGM_LISTNAMES
: This returns an array of node descriptions (as for NGM_NODEINFO) where each entry of the array describes a named node. All named nodes will be described.

NGM_LISTNODES
: This is the same as NGM_LISTNAMES except that all nodes are listed regardless of whether they have a name or not.

NGM_LISTTYPES
: This returns a list of all currently installed **netgraph** types.

NGM_TEXT_STATUS
: The node may return a text formatted status message. The status information is determined entirely by the node type. It is the only ``generic'' message that requires any support within the node itself and as such the node may elect to not support this message. The text response must be less than NG_TEXTRESPONSE bytes in length (presently 1024). This can be used to return general status information in human readable form.

NGM_BINARY2ASCII
: This message converts a binary control message to its ASCII form. The entire control message to be converted is contained within the arguments field of the NGM_BINARY2ASCII message itself. If successful, the reply will contain the same control message in ASCII form. A node will typically only know how to translate messages that it itself understands, so the target node of the NGM_BINARY2ASCII is often the same node that would actually receive that message.

NGM_ASCII2BINARY
: The opposite of NGM_BINARY2ASCII. The entire control message to be converted, in ASCII form, is contained in the arguments section of the NGM_ASCII2BINARY and need only have the *flags*, *cmdstr*, and *arglen* header fields filled in, plus the NUL-terminated string version of the arguments in the arguments field. If successful, the reply contains the binary version of the control message.

Flow Control Messages

In addition to the control messages that affect nodes with respect to the graph, there are also a number of *flow control* messages defined. At present these are *not* handled automatically by the system, so nodes need to handle them if they are going to be used in a graph utilising flow control, and will be in the likely path of these messages. The default action of a node that does not understand these messages should be to pass them onto the next node. Hopefully some helper functions will assist in this eventually. These messages are also defined in <*netgraph/ng_message.h*> and have a separate cookie NG_FLOW_COOKIE to help identify them. They will not be covered in depth here.

INITIALIZATION

The base **netgraph** code may either be statically compiled into the kernel or else loaded dynamically as a KLD via kldload(8). In the former case, include

options NETGRAPH

in your kernel configuration file. You may also include selected node types in the kernel compilation, for example:

options NETGRAPH
options NETGRAPH_SOCKET
options NETGRAPH_ECHO

Once the **netgraph** subsystem is loaded, individual node types may be loaded at any time as KLD modules via kldload(8). Moreover, **netgraph**

knows how to automatically do this; when a request to create a new node of unknown type *type* is made, **netgraph** will attempt to load the KLD module *ng_*<*type*>*.ko*.

Types can also be installed at boot time, as certain device drivers may want to export each instance of the device as a **netgraph** node.

In general, new types can be installed at any time from within the kernel by calling **ng_newtype**(), supplying a pointer to the type's *struct ng_type*

structure.

The **NETGRAPH_INIT**() macro automates this process by using a linker set.

EXISTING NODE TYPES

Several node types currently exist. Each is fully documented in its own man page:

SOCKET
: The socket type implements two new sockets in the new protocol domain PF_NETGRAPH. The new sockets protocols are NG_DATA and NG_CONTROL, both of type SOCK_DGRAM. Typically one of each is associated with a socket node. When both sockets have closed, the node will shut down. The NG_DATA socket is used for sending and receiving data, while the NG_CONTROL socket is used for sending and receiving control messages. Data and control messages are passed using the sendto(2) and recvfrom(2) system calls, using a *struct sockaddr_ng* socket address.

HOLE
: Responds only to generic messages and is a ``black hole'' for data. Useful for testing. Always accepts new hooks.

ECHO
: Responds only to generic messages and always echoes data back through the hook from which it arrived. Returns any non-generic messages as their own response. Useful for testing. Always accepts new hooks.

TEE
: This node is useful for ``snooping''. It has 4 hooks: *left*, *right*, *left2right*, and *right2left*. Data entering from the *right*

is passed to the *left* and duplicated on *right2left*, and data entering from the *left* is passed to the *right* and duplicated on *left2right*. Data entering from *left2right* is sent to the *right*

and data from *right2left* to *left*.

RFC1490 MUX
: Encapsulates/de-encapsulates frames encoded according to RFC 1490. Has a hook for the encapsulated packets (*downstream*) and one hook for each protocol (i.e., IP, PPP, etc.).

FRAME RELAY MUX
: Encapsulates/de-encapsulates Frame Relay frames. Has a hook for the encapsulated packets (*downstream*) and one hook for each DLCI.

FRAME RELAY LMI
: Automatically handles frame relay ``LMI'' (link management interface) operations and packets. Automatically probes and detects which of several LMI standards is in use at the exchange.

TTY
: This node is also a line discipline. It simply converts between

mbuf frames and sequential serial data, allowing a TTY to appear as a **netgraph** node. It has a programmable ``hotkey'' character.

ASYNC
This node encapsulates and de-encapsulates asynchronous frames according to RFC 1662. This is used in conjunction with the TTY node type for supporting PPP links over asynchronous serial lines.

ETHERNET
This node is attached to every Ethernet interface in the system. It allows capturing raw Ethernet frames from the network, as well as sending frames out of the interface.

INTERFACE
This node is also a system networking interface. It has hooks representing each protocol family (IP, AppleTalk, IPX, etc.) and appears in the output of ifconfig(8). The interfaces are named ``ng0'', ``ng1'', etc.

ONE2MANY
This node implements a simple round-robin multiplexer. It can be used for example to make several LAN ports act together to get a higher speed link between two machines.

Various PPP related nodes
There is a full multilink PPP implementation that runs in **netgraph**. The *net/mpd* port can use these modules to make a very low latency high capacity PPP system. It also supports PPTP VPNs using the PPTP node.

PPPOE
A server and client side implementation of PPPoE. Used in conjunction with either ppp(8) or the *net/mpd* port.

BRIDGE
This node, together with the Ethernet nodes, allows a very flexible bridging system to be implemented.

KSOCKET
This intriguing node looks like a socket to the system but diverts all data to and from the **netgraph** system for further processing. This allows such things as UDP tunnels to be almost trivially implemented from the command line.

Refer to the section at the end of this man page for more nodes types.

NOTES

Whether a named node exists can be checked by trying to send a control message to it (e.g., NGM_NODEINFO). If it does not exist, ENOENT will be returned.

All data messages are *mbuf chains* with the M_PKTHDR flag set.

Nodes are responsible for freeing what they allocate. There are three exceptions:

1. *Mbufs* sent across a data link are never to be freed by the sender. In the case of error, they should be considered freed.

2. Messages sent using one of **NG_SEND_MSG_***() family macros are freed by the recipient. As in the case above, the addresses associated with the message are freed by whatever allocated them so the recipient should copy them if it wants to keep that information.

3. Both control messages and data are delivered and queued with a **netgraph** *item*. The item must be freed using **NG_FREE_ITEM**(*item*) or passed on to another node.

FILES

<netgraph/netgraph.h>
Definitions for use solely within the kernel by **netgraph** nodes.

<netgraph/ng_message.h>
Definitions needed by any file that needs to deal with **netgraph** messages.

<netgraph/ng_socket.h>
Definitions needed to use **netgraph** *socket* type nodes.

<netgraph/ng_><*type*>.h
Definitions needed to use **netgraph** *type* nodes, including the type cookie definition.

/boot/kernel/netgraph.ko
The **netgraph** subsystem loadable KLD module.

/boot/kernel/ng_<*type*>.ko
Loadable KLD module for node type *type*.

src/sys/netgraph/ng_sample.c
Skeleton **netgraph** node. Use this as a starting point for new node types.

USER MODE SUPPORT

There is a library for supporting user-mode programs that wish to interact with the **netgraph** system. See netgraph(3) for details.

Two user-mode support programs, ngctl(8) and nghook(8), are available to assist manual configuration and debugging.

There are a few useful techniques for debugging new node types. First, implementing new node types in user-mode first makes debugging easier. The *tee* node type is also useful for debugging, especially in conjunction with ngctl(8) and nghook(8).

Also look in */usr/share/examples/netgraph* for solutions to several common networking problems, solved using **netgraph**.

SEE ALSO

socket(2), netgraph(3), ng_async(4), ng_atm(4), ng_atmllc(4), ng_bluetooth(4), ng_bpf(4), ng_bridge(4), ng_bt3c(4), ng_btsocket(4), ng_cisco(4), ng_device(4), ng_echo(4), ng_eiface(4), ng_etf(4), ng_ether(4), ng_fec(4), ng_frame_relay(4), ng_gif(4), ng_gif_demux(4), ng_h4(4), ng_hci(4), ng_hole(4), ng_hub(4), ng_iface(4), ng_ip_input(4), ng_ksocket(4), ng_l2cap(4), ng_l2tp(4), ng_lmi(4), ng_mppc(4), ng_netflow(4), ng_one2many(4), ng_ppp(4), ng_pppoe(4), ng_pptpgre(4), ng_rfc1490(4), ng_socket(4), ng_split(4), ng_sppp(4), ng_sscfu(4), ng_sscop(4), ng_tee(4), ng_tty(4), ng_ubt(4), ng_UI(4), ng_uni(4), ng_vjc(4), ng_vlan(4), ngctl(8), nghook(8)

HISTORY

The **netgraph** system was designed and first implemented at Whistle Communications, Inc. in a version of FreeBSD 2.2 customized for the Whistle InterJet. It first made its debut in the main tree in FreeBSD 3.4.

AUTHORS

Julian Elischer <julian@FreeBSD.org>, with contributions by Archie Cobbs <archie@FreeBSD.org>.