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 \texttt{ngctl(8)} command in action
Visualizing NETGRAPH systems

Details of frequently used NETGRAPH nodes
\texttt{ng\_ether, ng\_eiface}
\texttt{ng\_socket, ng\_ksocket}
\texttt{ng\_bridge, ng\_vlan, ng\_tee}
\texttt{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 \texttt{ng\_ipfw, ng\_tag, and ng\_bpf}
WAN Bridge using \texttt{ng\_bridge, ng\_eiface, ng\_ksocket, and OpenVPN or IPSEC}
\texttt{mpd5}: NETGRAPH based implementation of the multi-link PPP protocol for FreeBSD

Guidelines for implementing one’s own NETGRAPH node
Understanding \texttt{mbuf’s}
Essential node type methods
\texttt{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

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.

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)
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 with 10 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 atm11c 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 12cap 12tp mppc netflow one2many pptpgre split sppp ssafu sscoop ubt uni vlans

car ccatm deflate ipfw nat patch predl source sync_ar sync_sr
tag tcpmmms

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.
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
  \[ \text{nodename}: \quad (\text{for \ example, em0:}) \]
- 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:

![Diagram showing the location of NETGRAPH hooks in FreeBSD 4.x kernel](http://people.freebsd.org/~julian/layer2c.pdf)
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

```bash
# 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: ngctl199971   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
  ```bash
  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
gctl 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

```
em0:
ether [1]:
lower
right
T:
tee [5b]:
right2left
left2right
MyHookR2L
MyHookL2R
MyNode:
hole [5e]:
```
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

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

```bash
# 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";
    }
};

$ dot -Tpng ngctl.dot > ngctl.png
```

Snoop Example Revisited

```bash
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}:
```
**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

```bash
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

```bash
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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>ID</th>
<th>Num hooks</th>
</tr>
</thead>
<tbody>
<tr>
<td>vlan</td>
<td>vlan</td>
<td>00000038</td>
<td>3</td>
</tr>
<tr>
<td>local</td>
<td>peer</td>
<td>peer</td>
<td>peer</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>vlan123</td>
<td>eiface</td>
<td>0000003c</td>
<td>ether</td>
</tr>
<tr>
<td>nomatch</td>
<td>ether</td>
<td>0000000b</td>
<td>upper</td>
</tr>
<tr>
<td>downstream</td>
<td>ether</td>
<td>0000000b</td>
<td>lower</td>
</tr>
</tbody>
</table>
```
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,

gestats, 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

```sh
#!/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
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**

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.

```bash
kldload ng_ipfw; kldload ng_bpf; kldload ng_tag
ngctl mkpeer ipfw: bpf 41 ipfw
ngctl name ipfw:41 dcbpf
gctl mkpeer dcbpf: tag matched th1
ngctl name dcbpf:matched ngdc
grep MTAG_IPFW /usr/include/netinet/ip_fw
#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=0jf=0 k=8192 } ... ]

ipfw add 100 netgraph 41 tcp from any to any 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**

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.

```bash
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
gctl 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=0jf=0 k=40 } { code=21 jt=0jf=3 k=9299 } { code=32 jt=0jf=0 k=42 } { code=21 jt=0jf=1 k=170173500 } { code=6 jt=0jf=0 k=8192 } { code=6 jt=0jf=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**

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.

```bash
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
gctl 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=0jf=0 k=40 } { code=21 jt=0jf=3 k=9299 } { code=32 jt=0jf=0 k=42 } { code=21 jt=0jf=1 k=170173500 } { code=6 jt=0jf=0 k=8192 } { code=6 jt=0jf=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
```
MPD  Champion of netgraph(3) User Library

```
ng_async
ng_bpf
ng_car
ng_deflate
ng_ether
ngiface
ng_ksocket
ng_l2tp
ng_mppc
ngnat
ng_netflow
ng_ppp
ng_pppoede
ng_pptpgre
ng_pred1
ng_socket
ng_tcmid
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 asynchronous 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 way as asynchronous 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
12tp_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 12tp
set 12tp self 10.10.10.11 1701
set 12tp peer 10.10.10.10 1701
set 12tp enable outcall
set 12tp hostname slave
set 12tp secret MySecret
set link max-redial 0
set link action bundle B

12tp_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 12tp
set 12tp self 10.10.10.11 1701
set 12tp peer 10.10.10.10 1701
set 12tp enable outcall
set 12tp hostname slave
set 12tp 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

```c
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,
};
```

Custom Node `ng_sample.c`

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

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

```c
struct foobar *f;

if (m->m_len < sizeof(*f) && (m = m_pullup(m, sizeof(*f))) == NULL) {
    NG_FREE_META(meta);
    return (ENOBUSFS);
}

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

Debugging a netgraph node

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/18-2%20%20%206WINDGate%20Architecture%20Overview%20v1.0.pdf

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

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://www.youtube.com/watch?v=wh09MnPd5Y (MeetBSD talk, July 2010)
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

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

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

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

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 or 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``.

![Tee node type](image)

**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).

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

![Interface node type](image)

**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.

![TTY node type](image)

**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.

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

![Socket node type](image)
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:

```
node1
  hook1a hook2b

node2
  hook3a hook4b
```

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 "..:ihook3a.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 networks, using UDP packets 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):
# This script sets up a virtual point-to-point WAN link between two networks, using UDP packets 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):
# This script sets up a virtual point-to-point WAN link between two networks, using UDP packets 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):
# This script sets up a virtual point-to-point WAN link between two networks, using UDP packets 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):

LOC_INTERIOR_IP=192.168.1.1
LOC_EXTERNAL_IP=192.168.1.1
REM_INTERIOR_IP=192.168.2.1
REM_EXTERNAL_IP=192.168.2.1
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 take a look at it. We see that it has a name `"ngctl852"` 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 .
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>
Description: Create and connect a new node to the node at "path"
```

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: 2
  local hook Peer name Peer type Peer ID Peer hook
    """""""""""""""""""""""""""""""
  right  npt1652 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
```

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
```

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: npt1652 Type: socket ID: 00000045 Num hooks: 1
Name: xpt0 Type: ether ID: 00000002 Num hooks: 0
Name: xpt1 Type: ether ID: 00000001 Num hooks: 0
```

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...

```
+ msg mytee: getstats
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                          ... ....
Rec'd response "getstats" (1) from "mytee:"
```

Now let's shut down (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  npt1652 socket 00000045 myhook
```

```
+ help msg
Usage: msg path command [args ... ]
```

The reply is simply an ASCII version of the structure `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
+ mg .myhook2 connect inet/127.0.0.1:13
Rec'd data packet on hook "myhook":
0000: 54 75 65 20 46 65 62 20 00 00 00 00 00 00 00 00  Tue Feb 1 11:02
0010: 3a 32 38 20 32 00 00 00 00 00 00 00 00 00 00 00  12:38...

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 freed 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:

```c
/* 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_disconnect_t *disconnect;    /* notify on disconnect */
    const struct ng_cmdlist *cmdlist;    /* commands we can convert */
    u_int32_t 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 nod_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.

```c
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.

**Default action:** Just calls ng_make_node_common().

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

```c
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:** 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.

```c
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()).

**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.

```c
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:** Calls ng_make_node_common().

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

```c
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).

```c
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 always accepted.

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

```c
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 \( m == \text{NULL} \) (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.

```c
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.

```c
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->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.

```c
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 *
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_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. The message is queued and delivered later at `splnet()`. The `msg` and `meta` information will be queued and delivered later at `splnet()`. This is equivalent to calling `ng_send_data()`. The mbuf and meta-information will be queued and delivered later at `splnet()`.

**NG_SEND_DATAQ(error, hook, m, meta)**

What it does: Delivers the mbuf and associated meta-data out the 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 function 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_DATAQ()` macro instead).

**NG_SEND_DATAQ(hook_p hook, struct mbuf *m, meta_p meta);**

What it does: Calls `ng_send_data()` and `ng_queue_msg()`.

**int ng_send_data(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_msg(node_p node, 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_queue_msg()`, 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 node, struct ng_mesg *msg, const char *address, struct ng_mesg **resp);**

What it does: Delivers 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_send_msg(node_p here, struct ng_mesg *msg, const char *address);**

What it does: Delivers 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_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_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() 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_TEE_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 `NGM_TEE_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 `ng_t_rmnode()`.

- Also in `ng_t_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 `ng_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 structs `ng_cmdlist`. 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 mbufs work and are used.
- All data packets must be packet header mbufs, 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:

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

- 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 BF.
- 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), pp(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 pp(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
NGCTL(8) FreeBSD System Manager's Manual

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 immedi-
ately.

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 filename
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>
cconnect 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
clist Show information about all nodes
cmap Create and connect a new node to the node at <path>
mess Send a netgraph control message to the node at <path>
cname Assign name <name> to the node at <path>
cread Read and execute commands from a file
crmhook Disconnect hook "hook" of the node at <path>
cshutdown Shutdown the node at <path>
cstatus Get human readable status information from the node at <path>
types Show information about all installed node types
cwrite Send a data packet down the hook named by "hook"
cquit Exit program

cSome commands have aliases, e.g., "list" 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(8), netgraph(4), ngctl(8)

HISTORY
The netgraph system was designed and first implemented at Whistle Commu-
ications, Inc. in a version of FreeBSD 2.2 customized for the Whistle
Interjet.

AUTHORS
Archie Cobbs <archie@whistle.com>

BUGS
None known.

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

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

HISTORY
The netgraph system was designed and first implemented at Whistle Commu-
ications, Inc. in a version of FreeBSD 2.2 customized for the Whistle
Interjet.

AUTHORS
Archie Cobbs <archie@whistle.com>
A hook may supply override 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, hooks are used for flow control and link management purposes are defined by the base node, but may also be overloaded by other node types if desired. In object-oriented language, types are classes, and nodes are instances of those classes.

Netgraph is a flexible way of combining protocol and link level drivers. It is a reasonably fast, kernel-based implementation. 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).

Hooks are connected together in arbitrary order, the edges forming a graph. Each node decides how to handle and process data, and local messages flow along the edges in the graph. Along with data, nodes can also receive user-specific control messages. Those messages are generally not understood by other nodes, though they may be interpreted by the hook that was used to deliver them. Nodes are not required to understand the types of messages being sent to them in this manner.

Mon 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 purposes. Nodes are not required to support these conversions.

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). 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 node type has a corresponding typecookie. Messages are delivered by making function calls, not by queuing packets. The node that receives a message must determine how to process it. In general, a hook 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 (otherwise known as a type; e.g., NGключенчие) that is sufficient to identify the node. 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 can be used in two ways: they may be used to connect two hooks, or they may be used to define node types. Hooks can also be used to define node types. A hook is always connected to another hook. That is, a hook is always connected to another hook. Hooks are connected to one another only when they are defined as part of a node type. This allows nodes to be connected or disconnected in any order. Nodes are not required to support these conversions.
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 B's `receive data'' method. There are exceptions to this.

Node Methods

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 is the reader/writer semantic so that when there is a writer in the node, all other requests are queued, and while there are readers, a no reason to queue the data, the input method is called directly, as mentioned above.

A node may declare that all request should be considered as writers, or a writer only. For example, if device nodes are used, the node may wish to allow data to pass through without being read, such as the case for printer nodes. Of course, for data received by this function. The node is notified on which hook the item has arrived, and can use this information in its processing decision. The receiving node must send a request to the node the item was received from.

Interaction with Other Parts of the Kernel

It is possible for an infinite loop to occur if the graph contains a chain and reconnect its peers together, like the `ng_tee(4)` does.

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

Also, if node A wishes to make a data mbuf to neighboring node B, it calls the generic `netgraph B's `receive data'' method. There are exceptions to this.

A node may declare that all request should be considered as writers, or a writer only. For example, if device nodes are used, the node may wish to allow data to pass through without being read, such as the case for printer nodes. Of course, for data received by this function. The node is notified on which hook the item has arrived, and can use this information in its processing decision. The receiving node must send a request to the node the item was received from.

Interaction with Other Parts of the Kernel

It is possible for an infinite loop to occur if the graph contains a chain and reconnect its peers together, like the `ng_tee(4)` does.
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. (OC multiple freees 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_GET_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.

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() macro 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 item must free the message itself using the NG_FREE_M() 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() type N_MEMORY macro. 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_M() 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/rcvdata() 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.

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:

```
foo:hook1
[d80]:hook2
```

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]<--+(mux)<--[un-named ]
[ A ]   [ B ][dlci20]<--(mux)<-- [ C ]
[   ]   [ ][ typeof PPP ]
[ the](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 mix. 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. When B receives a frame on hook data, it decodes the frame relay header to determine the DLCI, and then forwards the uncrapped 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/msg_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.

- **Node**
  - **ID**: `ng_ID_t` (This property can be retrieved using the macro `NG_NODE_ID(node)`.)
  - **Name**: Optional globally unique name, NUL terminated string. If there is a value in here, it is the name of the node. (`NG_NODE_NAME(node)` 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.)
  - **Number of Hooks**: `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)` 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.

- **Hook**
  - **Type Cookie**: `NG_HOOK_TYPE(node)` will return this property, respectively.
  - **Node ID**: `NG_NODE_NUMHOOKS(node)` macro is used to retrieve this value.

**Netgraph Message Structure**

Control messages have the following structure:

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

struct ng_mesg {
  struct ng_msghdr {
    u_char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
    char   data[0];      /* Start of cmd/resp data */
  header;
    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 */
    char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
  header;
  struct ng_hook   *rethook; /* Hook that called this macro */
  }
header;
struct ng_hook {
  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 */
    char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
  header;
  char   data[0];      /* Start of cmd/resp data */
};
```

A hook dependent opaque cookie Anything of the pointer type can be placed here. The macros `NG_HOOK_SET_PRIVATE` and `NG_HOOK_GET_PRIVATE` set and retrieve this property, respectively.

- **Associate Node**: `NG_NODE_SET_PRIVATE(node, value)` and `NG_NODE_GET_PRIVATE(node)` set and retrieve this property, respectively.

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_RECVMSG(hook, fn)` macro finds the peer.

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.

- **Node**
  - **ID**: `ng_ID_t` (This property can be retrieved using the macro `NG_NODE_ID(node)`.)
  - **Name**: Optional globally unique name, NUL terminated string. If there is a value in here, it is the name of the node. (`NG_NODE_NAME(node)` 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.)
  - **Number of Hooks**: `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)` 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.

- **Hook**
  - **Type Cookie**: `NG_HOOK_TYPE(node)` will return this property, respectively.
  - **Node ID**: `NG_NODE_NUMHOOKS(node)` macro is used to retrieve this value.

**Netgraph Message Structure**

Control messages have the following structure:

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

struct ng_mesg {
  struct ng_msghdr {
    u_char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
    char   data[0];      /* Start of cmd/resp data */
  header;
    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 */
    char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
  header;
struct ng_hook {
  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 */
    char      cmdstr[NG_CMDSTRSIZ]; /* Cmd string (for debug) */
  header;
  char   data[0];      /* Start of cmd/resp data */
};
```
Structs are enclosed in double quotes and respect the normal C language backslash escapes. 

Strings are enclosed in double quotes.

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:

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

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

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

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

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.

Each module may define its own arbitrary types by providing the necessary routines to parse and unparse. ASCII forms defined for a specific module may also be converted to and from an equivalent ASCII form.

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 may also be converted to and from an equivalent ASCII form.

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.

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

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, the default value is usually zero; for string types, the empty string.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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.

Each type should have an include file that defines the commands, argument formats and command fields for its own messages. The typecookie must be a unique integer value. The typecookie is defined in the same header file as the command, the typecookie is a string version of the header field.

Arguments are represented by base 8, 10, or 16 numbers.

Awarding a cleanly detached module is not hard, but only takes a little time.

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

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.

Array elements and structure fields may be specified in any order. Elements in the same array must follow each other sequentially starting at index zero. An element may have an optional default value, which will work for any default value.

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 may also be converted to and from an equivalent ASCII form.
Once the netgraph subsystem is loaded, individual node types may be
loaded at any time as KLD modules via kldload(8). However, netgraph
knows how to automatically do this when a request to create a new node
is made, so the kernel module does not need to be rewritten in most
cases. (However, the module will still be a loadable module.)

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
structure.

The NGM_FOREACH macro automates this process by using a linker set.

**NGM_FOREACH**

```
foreach (node) { ...
}
```

This executes the same as NGM_FOREACH array, except that all nodes are
listed regardless of whether they have a name or not.

**NGM_FOREACH**

```
foreach (node) { ...
}
```

This returns a list of all currently installed netgraph types.

**Netgraph Structure**

The node structure contains a list of node descriptions (as for NET.SOFTWARE)
where each entry of the array describes a named node. All needed
options are also present.

**Netgraph Description**

`options NETGRAPH`

The node structure contains a list of node descriptions (as for NET.SOFTWARE)
where each entry of the array describes a named node. All needed
options are also present.

**Netgraph Flow Control**

`options NETGRAPH FLOW`.

There are a number of flow control messages defined in the
`netgraph/ng_message.h` header file. These messages are used to
handle communication between nodes and are typically sent and
received with the `sendto(2)` and `recvfrom(2)` system calls,
respectively. The messages include:

- **FLOW**
- **FLOW_ACCEPT**
- **FLOW_REJECT**
- **FLOW_TRAFFIC**

These messages are used to control the flow of data between nodes.

**Netgraph Initialization**

The library initializes the netgraph subsystem at boot time. This includes:

- Loading of KLD modules
- Initialization of node references
- Creation of node types

**Netgraph Types**

Various node types are currently installed. Each is fully documented in its
own manual page, and the following pages describe the defaults.

- **FLOW**
- **FLOW_ACCEPT**
- **FLOW_REJECT**
- **FLOW_TRAFFIC**

These messages are used to control the flow of data between nodes.

**Netgraph Options**

Options can be set in the configuration file to control the behavior of
netgraph nodes.

- **FLOW**
- **FLOW_ACCEPT**
- **FLOW_REJECT**
- **FLOW_TRAFFIC**

These messages are used to control the flow of data between nodes.

**Netgraph Flow Control**

The node structure contains a list of node descriptions (as for NET.SOFTWARE)
where each entry of the array describes a named node. All needed
options are also present.

**Netgraph Description**

`options NETGRAPH`
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 `sconfig()`. The interfaces are named `mg0`, `mg1`, 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/spd` port can use these modules to make a very low latency high capacity PPP system. It also supports PPP over IP using the PPTP node.

PPPOE A server and client side implementation of PPPoE. Used in conjunction with either `ppp(8)` or the `net/spd` 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 node types.

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.

SEE ALSO

* `ng_bluetooth(4)`
* `ng_bpf(4)`
* `ng_cisco(4)`
* `ng_echo(4)`
* `ng_eiface(4)`
* `ng_etf(4)`
* `ng:frame_relay(4)`
* `ng_gif(4)`
* `ng_h4(4)`
* `ng_hole(4)`
* `ng_hub(4)`
* `ng_iface(4)`
* `ng_ipinput(4)`
* `ng ISPs(4)`
* `ng_ksocket(4)`
* `ng_memcpy(4)`
* `ng_MSIP(4)`
* `ng_mplt(4)`
* `ng_mti(4)`
* `ng/pppe(4)`
* `ng_ppp(4)`
* `ng_ppp(4)`
* `ng_rfc1490(4)`
* `ng_socket(4)`
* `ng_split(4)`
* `ng_sppp(4)`
* `ng_sscfu(4)`
* `ng_sscop(4)`
* `ng_UI(4)`
* `ng_uni(4)`
* `ng_vjc(4)`

NOTES

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

All data messages are `mbuf` chains with the `M_PKTMRU` 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.

USER NODE 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, `nghook(8)` and `ngctl(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 `nghook(8)` and `ngctl(8)`.

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

AUTHORS

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

FreeBSD 8.2  May 25, 2008 FreeBSD 8.2