Network stack virtualization for FreeBSD 7.0

Marko Zec

zec@fer.hr

University of Zagreb

September 2007.

Tutorial outline

- Network stack virtualization concepts
- Management interface
- Application scenarios
- Implementation in FreeBSD 7.0 kernel

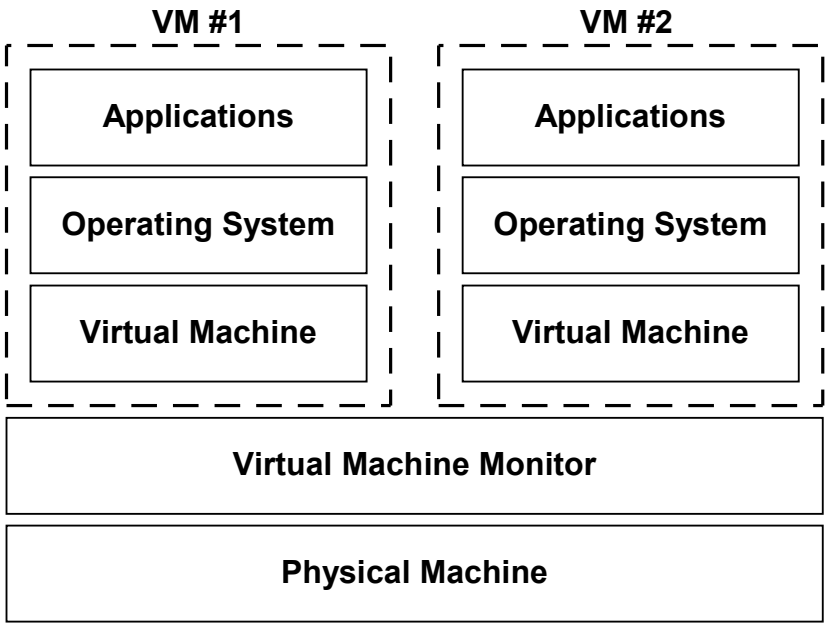
Tutorial outline

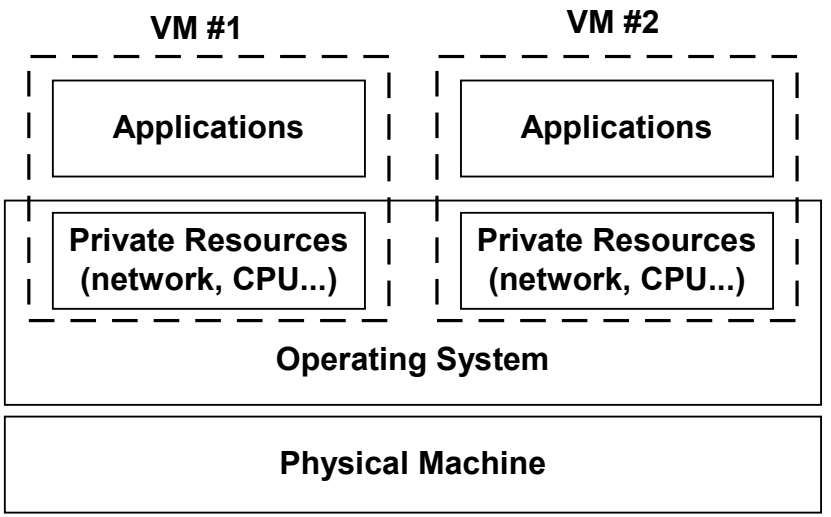
- Hands-on work encouraged!
- Bootable system image alternatives:
  - USB stick
  - LiveCD
  - VMWare player image
- Neither option will touch anything on your HD
- If already running a *recent* 7.0-CURRENT, you'll need to copy the kernel from the stick or CD, or compile your own

Tutorial outline

- Virtualized networking == preview technology
  - What we learn / try out today might look different in 3 months, particularly the management API
  - Your input may influence the final version of the virtualization framework
- Do not hesitate to ask questions!

Server virtualization: two sides of the spectrum





Strong isolation model

Independent OS instances

VM migration possible

Efficient resource utilization

No extra I/O overhead

Scaling

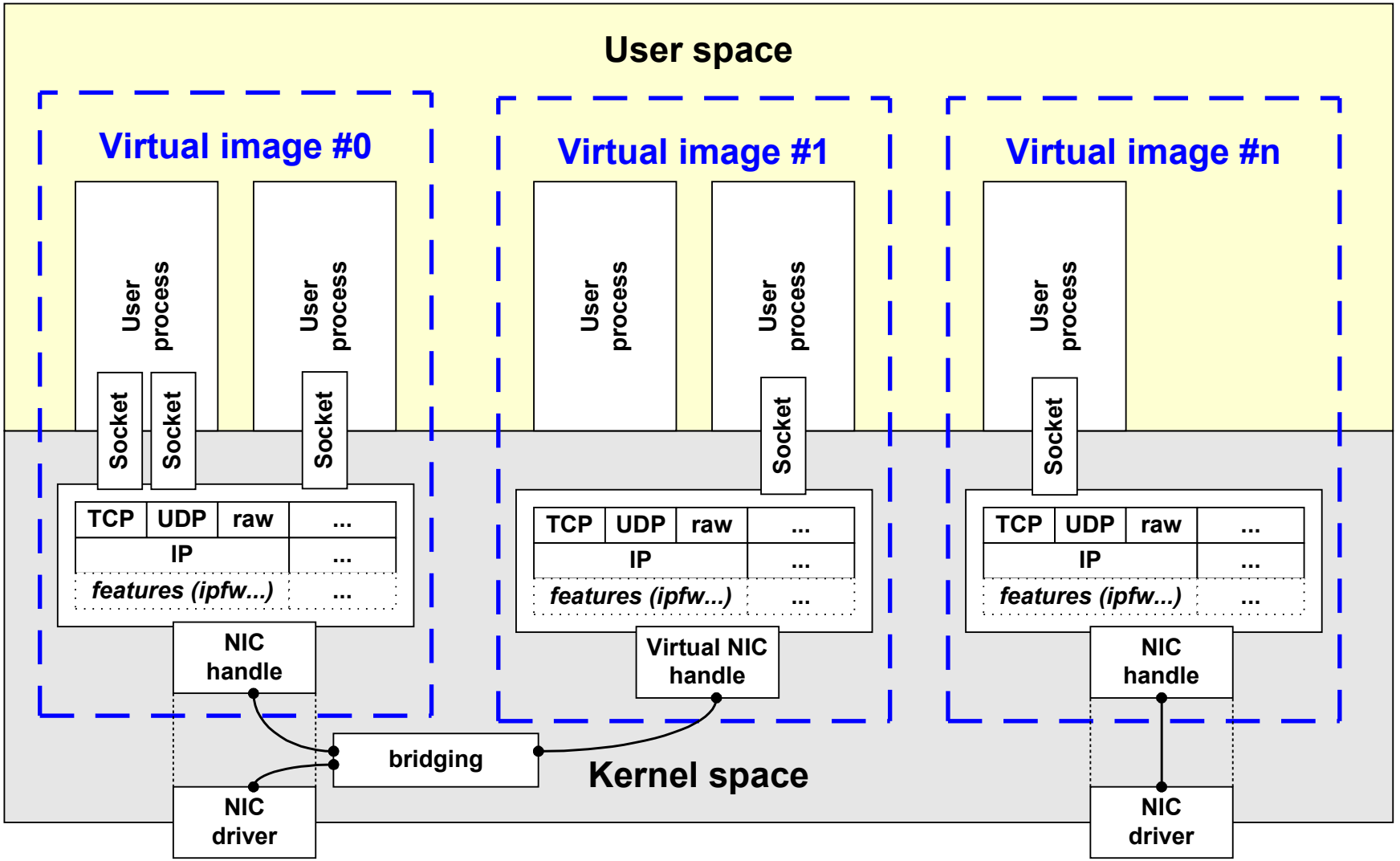
Motivation: the idea

- Traditional OS architecture
  - Support for only a single instance of network stack or protocol family within the kernel
  - *Jails*: first successful pseudo-virtualization framework
- Network stack virtualization (or *cloning)*
  - Multiple independent network stack *state* instances within a single kernel
  - Existing networking code paths and algorithms remain the same, but must be taught on how to operate on virtualized symbols / state

Applications: who needs this?

- Virtual hosting
  - Think of extending FreeBSD `jail` with its own independent network stack instance: multiple interfaces and IPv4 / IPv6 addresses, private routing table, IPFW / PF, dummynet, BPF, raw sockets etc. etc.
  - Anecdotal evidence: FreeBSD 4.11 based version in production use by some US ISPs
- VPN provisioning and monitoring
  - Support for overlapping IP addressing schemes
- Network simulation / emulation
  - Each network stack instance == an independent virtual node or router

Idea: replicate global networking state



Basic concepts

- Sockets
  - Permanently assigned to a stack instance at creation time
- Network interfaces
  - Belong to only one network stack instance at a time
  - Some interface types can be reassigned to other stacks
- User processes
  - Bound to only one stack at a time, can reassociate
  - Jail–style separation (reused existing jail code)

Implementation: API / ABI compatibility

- Userland to kernel: both API and ABI 100% preserved
  - Support for accessing the virtualized symbols added to the `kldsym` interface (`netstat` et. al.)
  - `sysctl` extensions to access virtualized state
- Within the kernel: internal APIs slightly modified
  - Hooks for creating / destroying vnet instances
  - Passing vnet context in function call graph
  - Storing vnet affinity with interfaces, sockets and more
  - Generally, no changes at device driver layer

History

- Initial implementation in 2002.
  - FreeBSD 4.7 (5.0-CURRENT wasn't sufficiently stable yet)
  - First presented at EuroBSDCon in November '02
- Focus on network simulation / emulation
- Tracked 4.x until 4.11-RELEASE
  - IPv6 virtualization done in cooperation with USC/ISI
  - IPSec virtualization code recently contributed by Boeing Airospace

(Re)implementation in 7.0-CURRENT

- What's different from 4.11 version:
  - Conditional compilation
  - Better support for kernel loadable modules
  - Code churn reduced by implicit vnet context propagation: argument lists to most networking functions remain unchanged
  - IPv6 support from day 0
  - Designed with SMP in mind from day 0
- Otherwise, no chances for including the changes into the main FreeBSD tree

Project status – FreeBSD 7.0-CURRENT

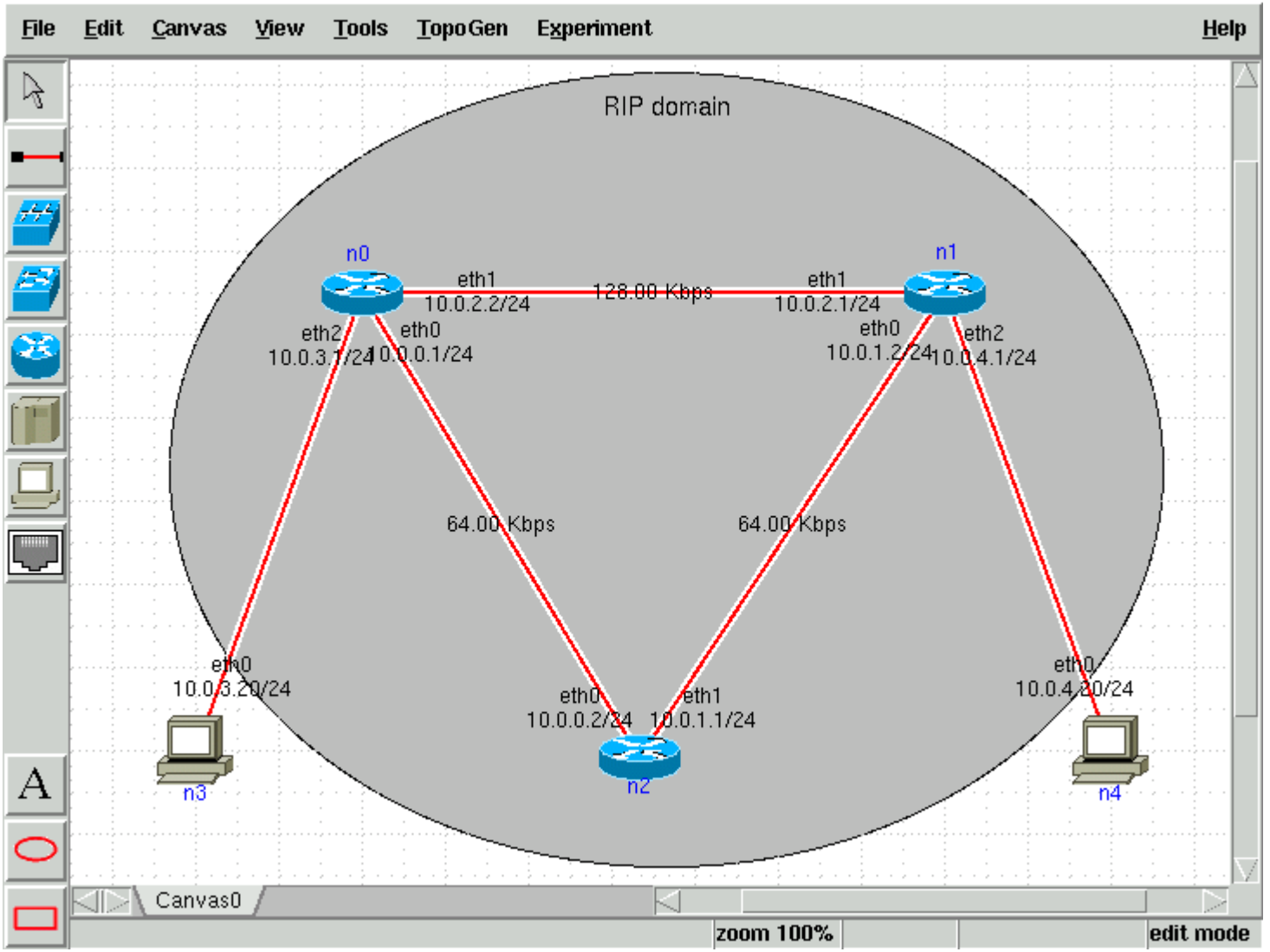
- Supported by NLNet and FreeBSD foundation
  - Started in August 2006, should have already finished...
- In sync with -CURRENT: p4 projects/vimage
  - Snap-in replacement kernel – no userspace changes!
  - http://imunes.tel.fer.hr/virtnet/ : snapshots (and CVSup)
- Reasonably stable already
  - Lots to be done: locking, management API & housekeeping
- Most important networking subsystems virtualized:
  - IPv4, IPv6, IPSec, NFS, IPFW / PF firewalls, BPF, raw / routing sockets...
- Outside the tree until 7.0-RELEASE, merging in 8.0?

Demo 1: network simulation / emulation

- Objectives:
  - Become familiar with stack virtualization concepts
- Steps:
  - Create a virtual network topology
  - Observe network configuration and processes in virtual nodes
  - Observe traffic within the virtual network
  - Observe rerouting around a link failure
  - Connect the virtual network with external world

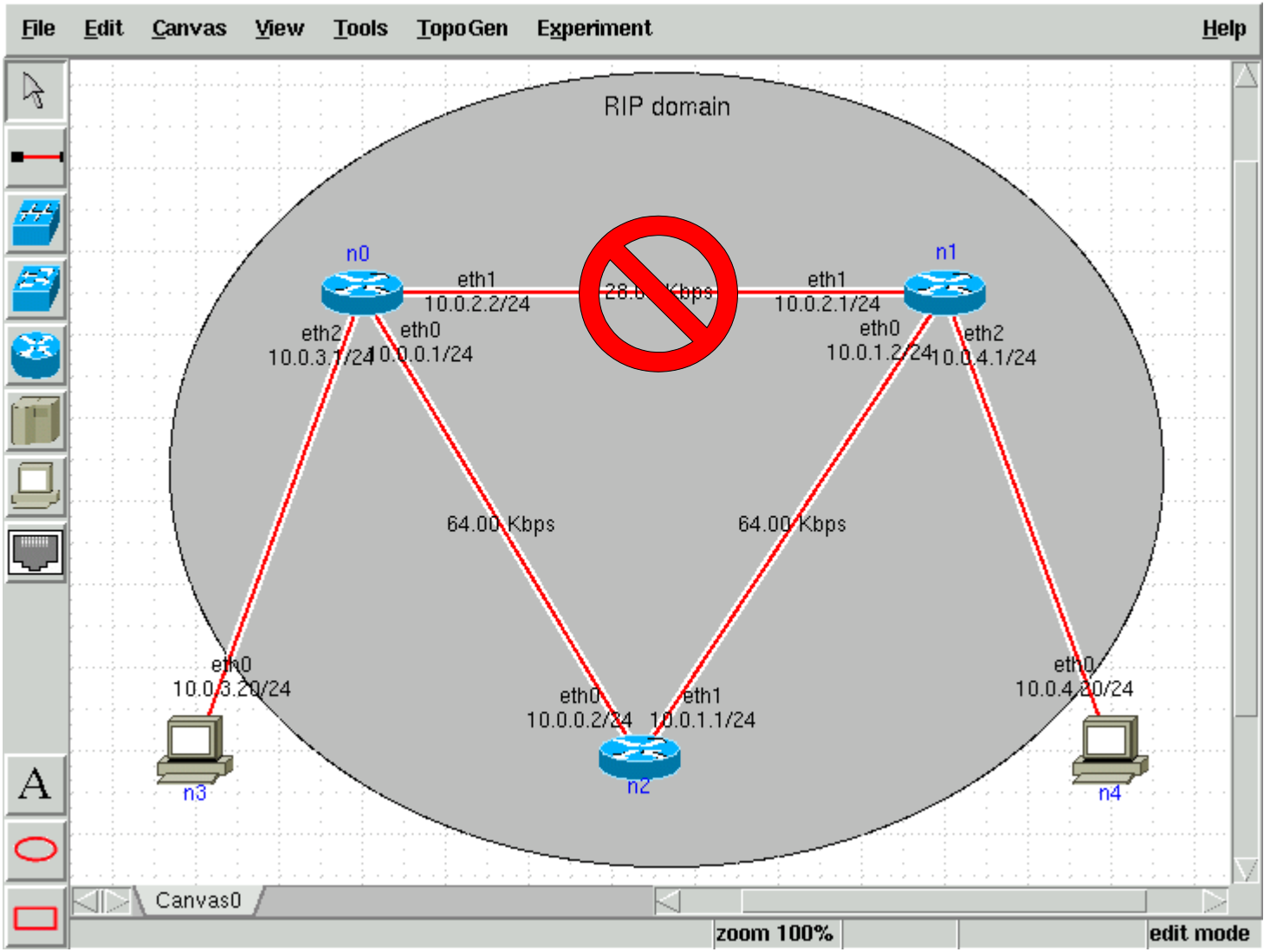
Demo 1: network simulation / emulation

- Nodes: virtual network stack instances
- Links: netgraph traffic shaping nodes



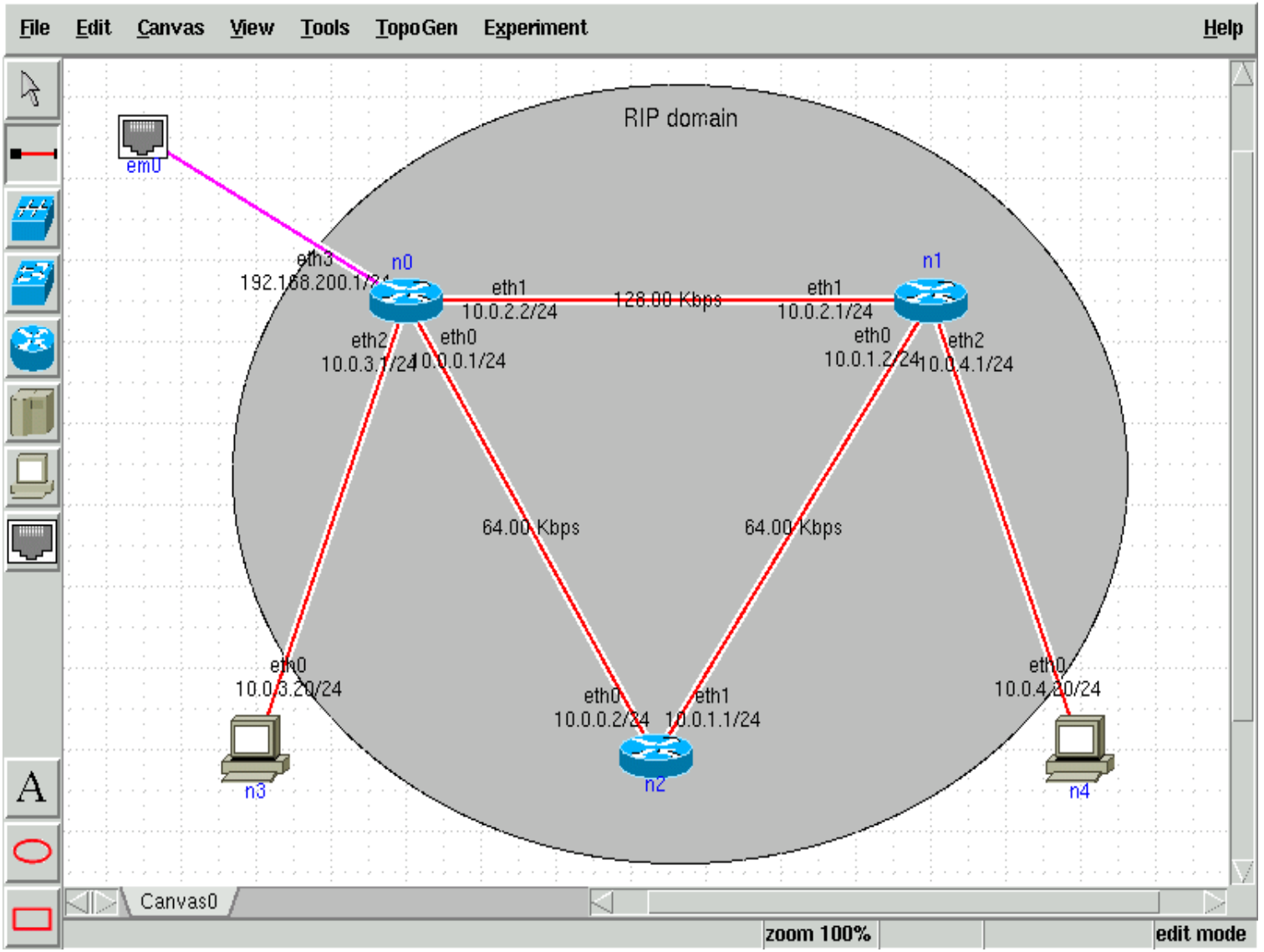
Rerouting: a link breaks

- Link emulation node configured to drop all packets
- ripd instances timeout on updates; reroute traffic



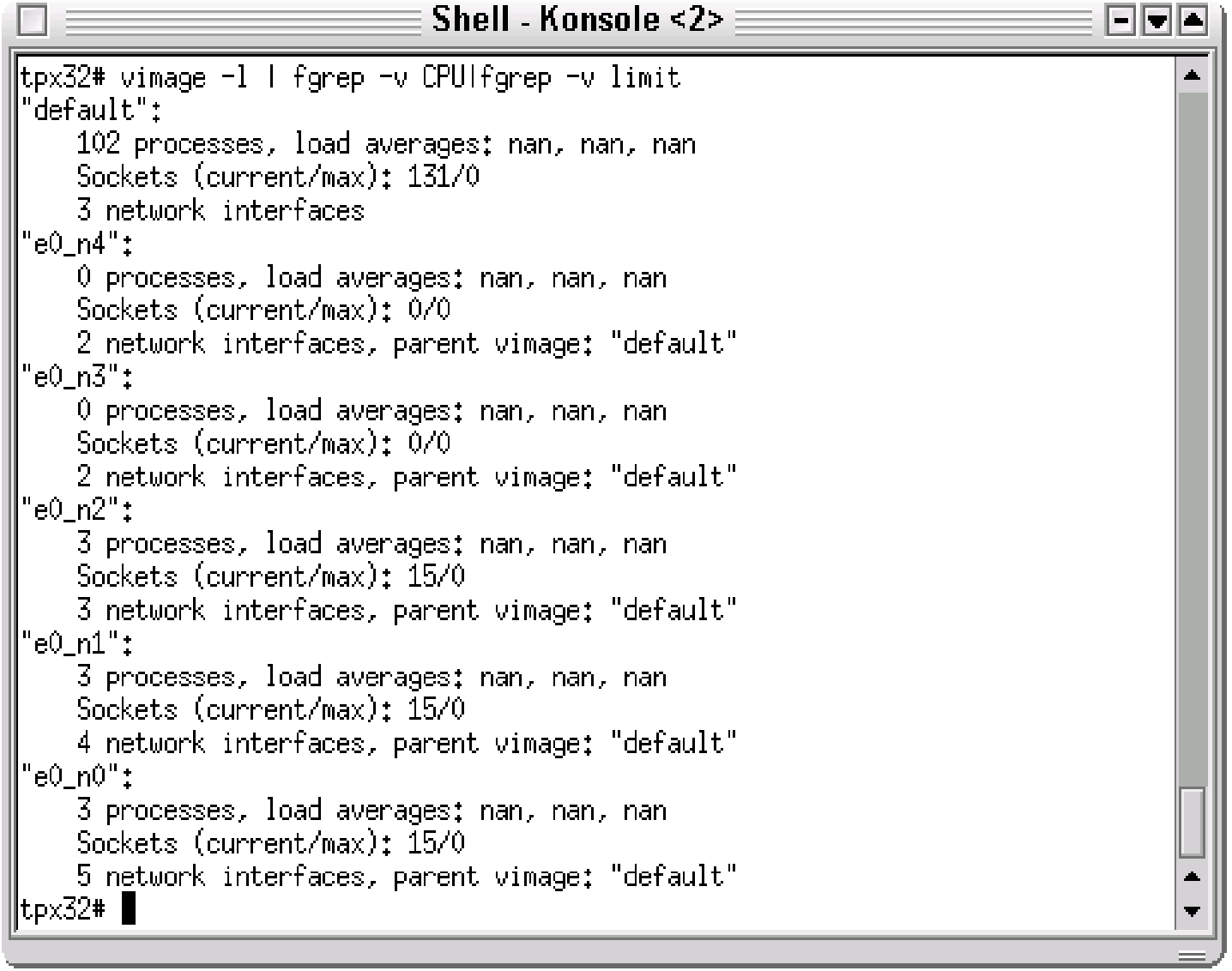
Connecting to the external world...

- Interface eth3 in n0 (192.168.200.1) connected to “em0” physical interface
- External traffic can enter virtual topology and vice versa



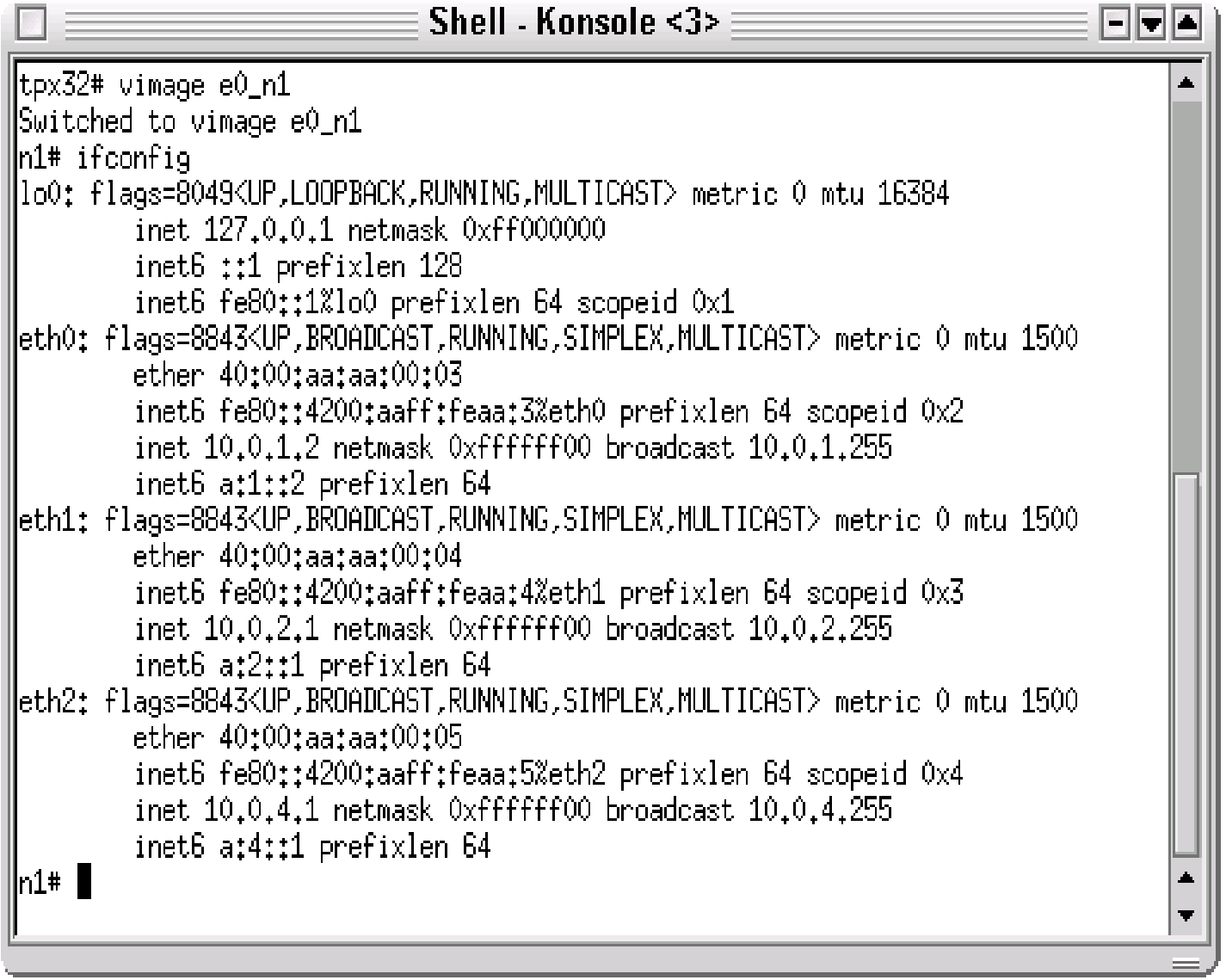
Under the hood: vimages...

- Virtual nodes correspond to *vimages*
- “default” vimage always present
- Can exist without running processes or open sockets!



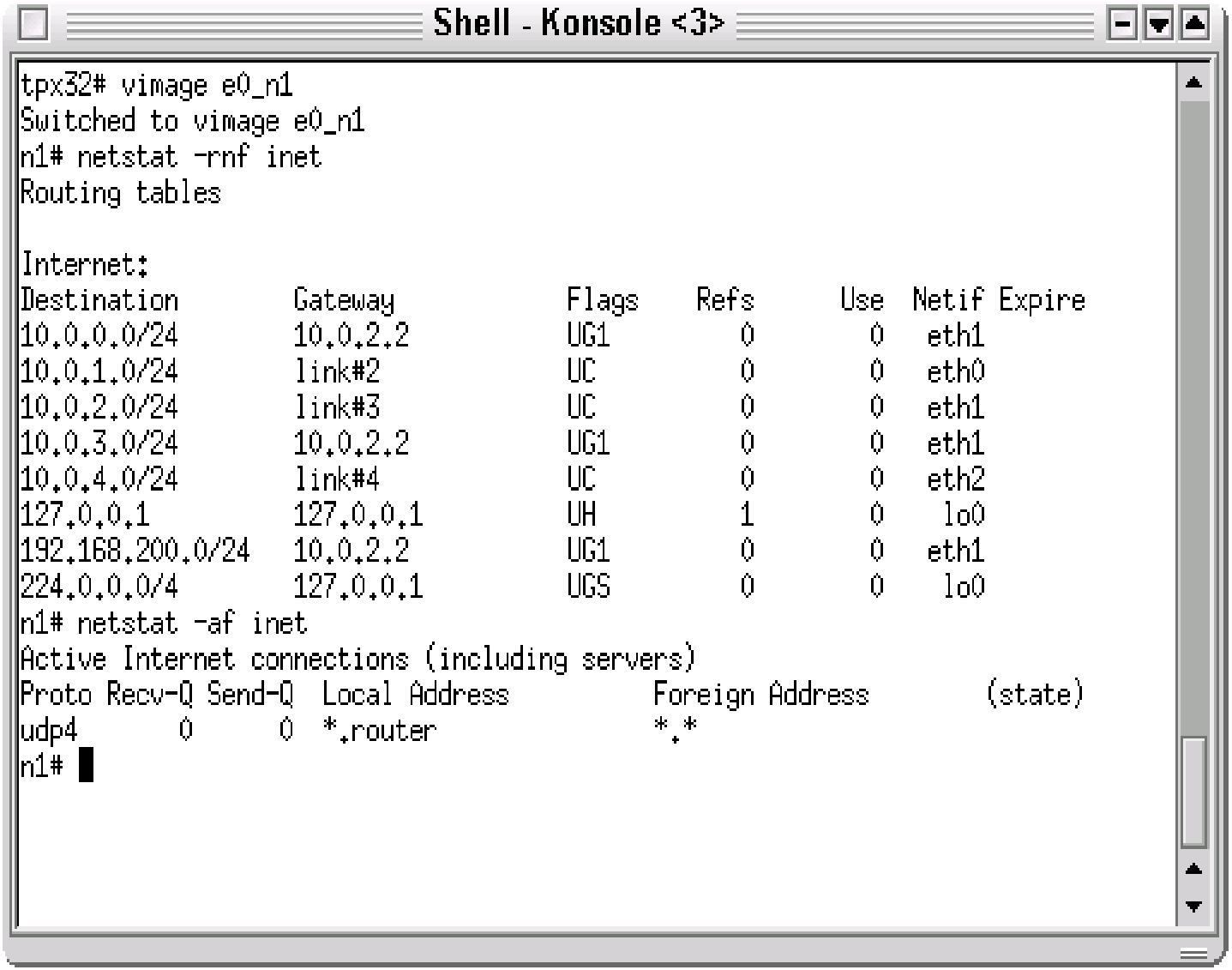
Under the hood: vimages...

- Each vimage maintains its fully private / independent interface list
- Each vimage has its own loopback interface instance
- Independent IPv4 / IPv6 address space



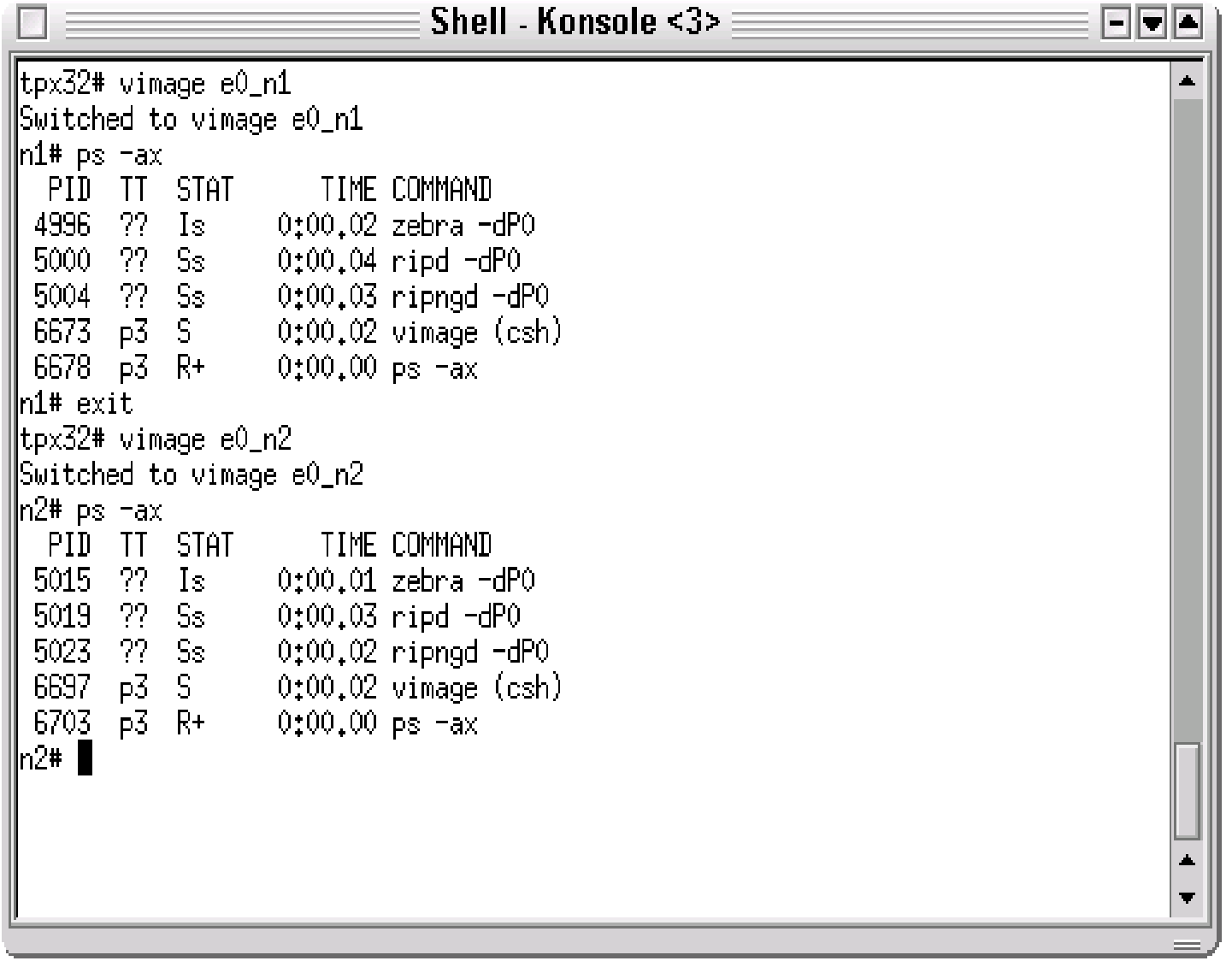
Under the hood: vimages...

- Each vimage maintains its private routing table
- Independent port / socket space
- Independent netgraph space!



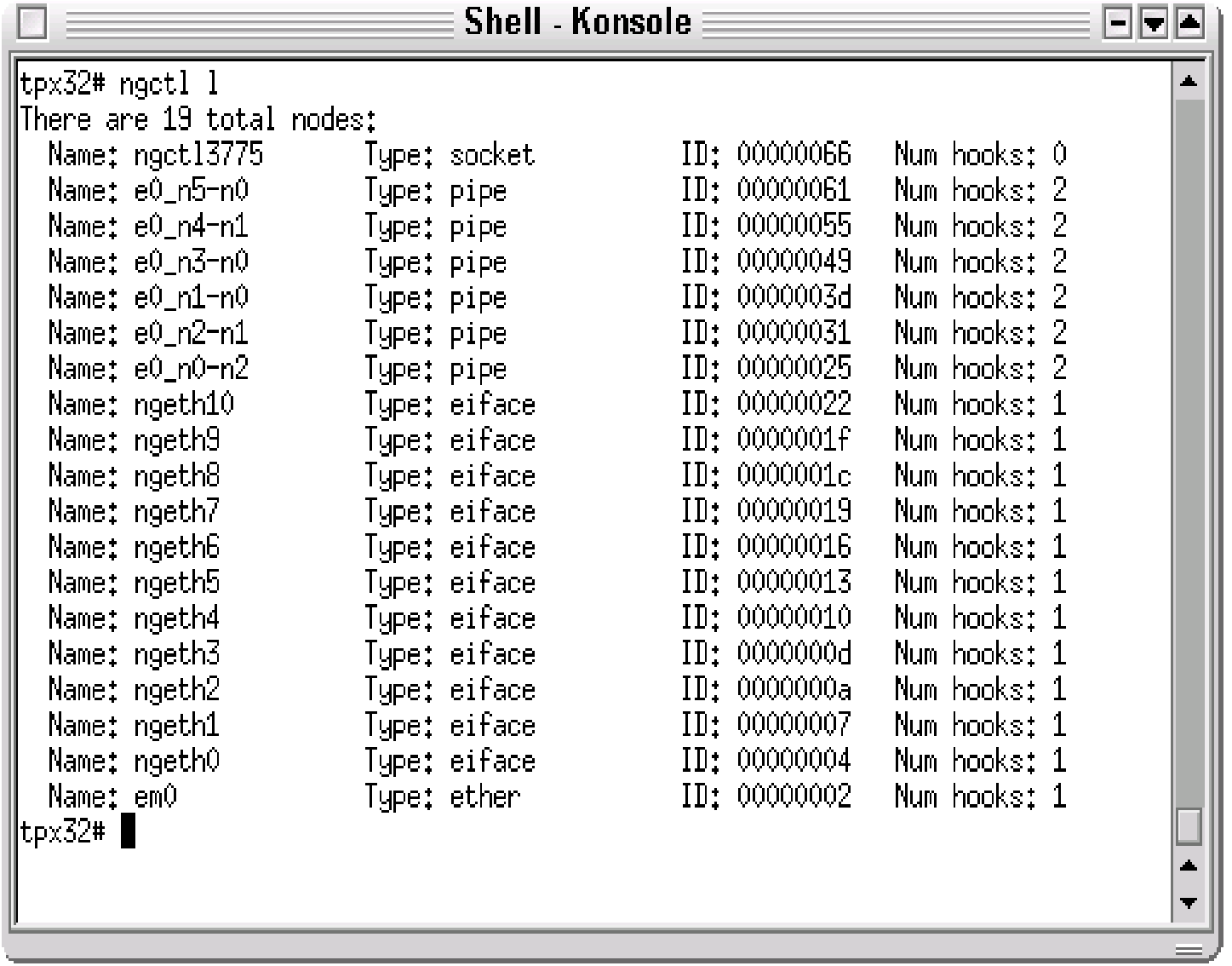
Under the hood: vimages...

- Each vimage has it own process group
- Code inherited/ shared with *jail* implementation
- Technically, process grouping independent from stack virtualization

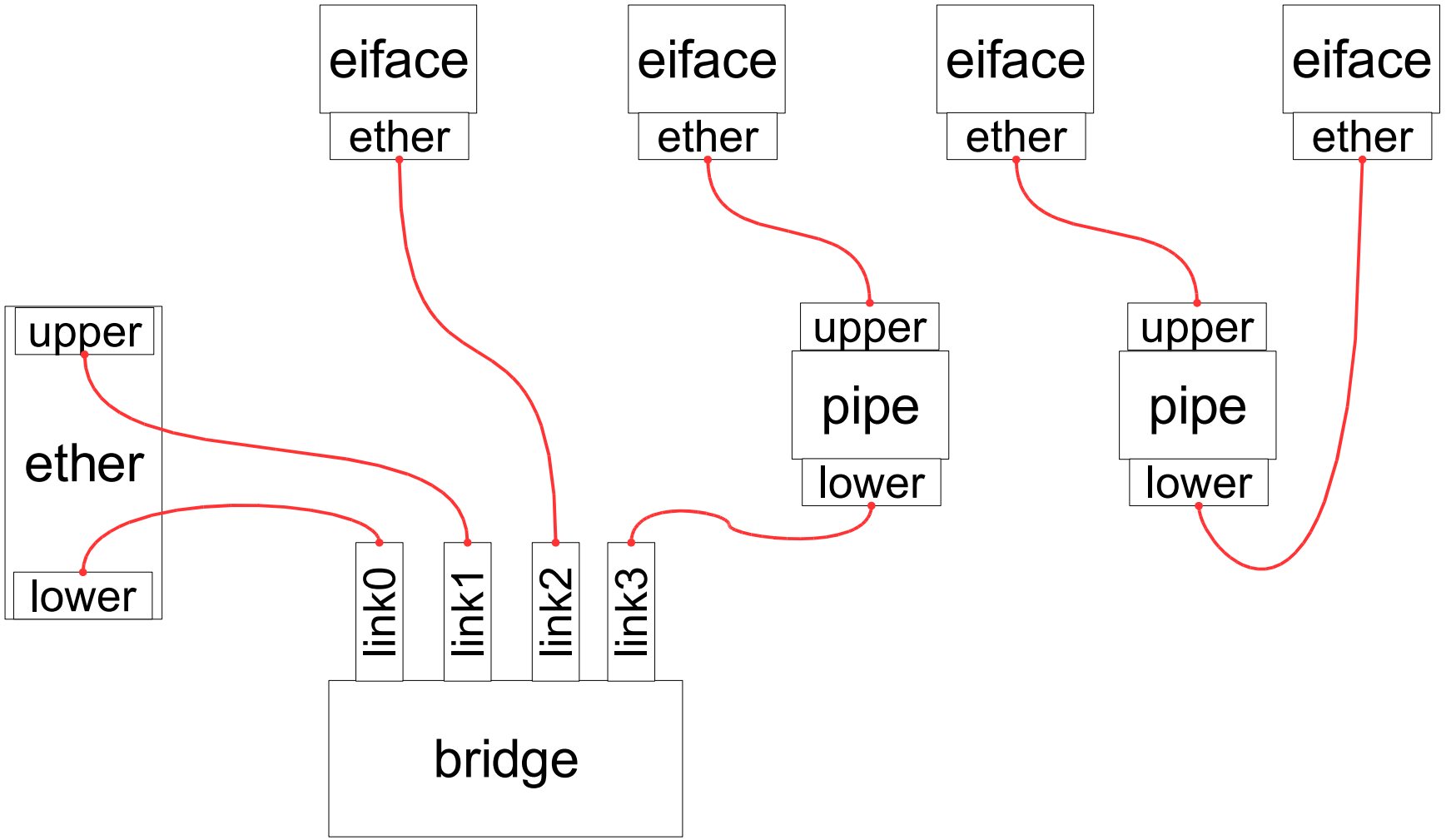


Under the hood: netgraph

- Virtual topology constructed by interconnecting netgraph *nodes*
- Connection points between nodes: *hooks*
- Hooks: bidirectional interfaces
- Data / control messages



Netgraph? Meet the nodes...



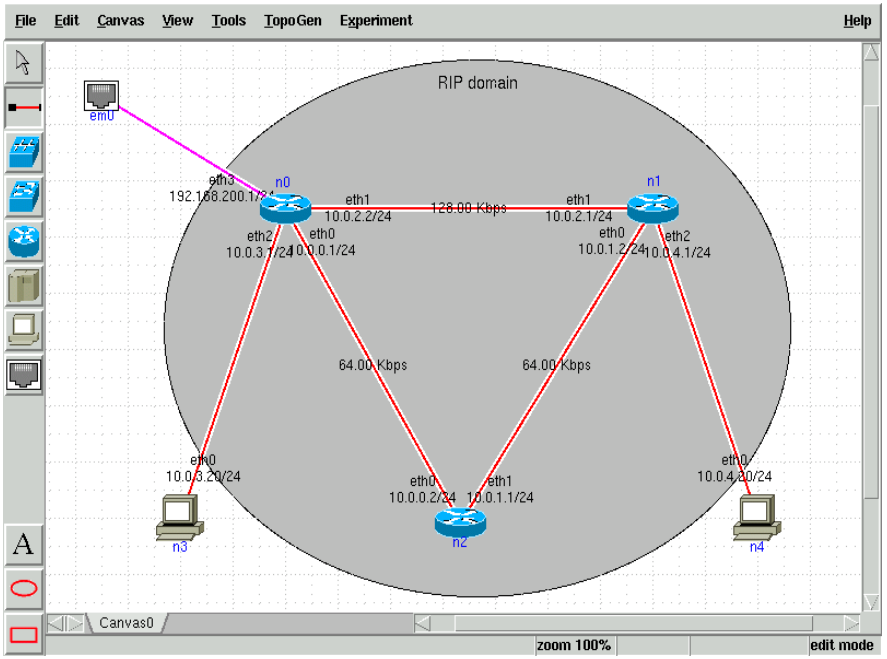
Netgraph node types

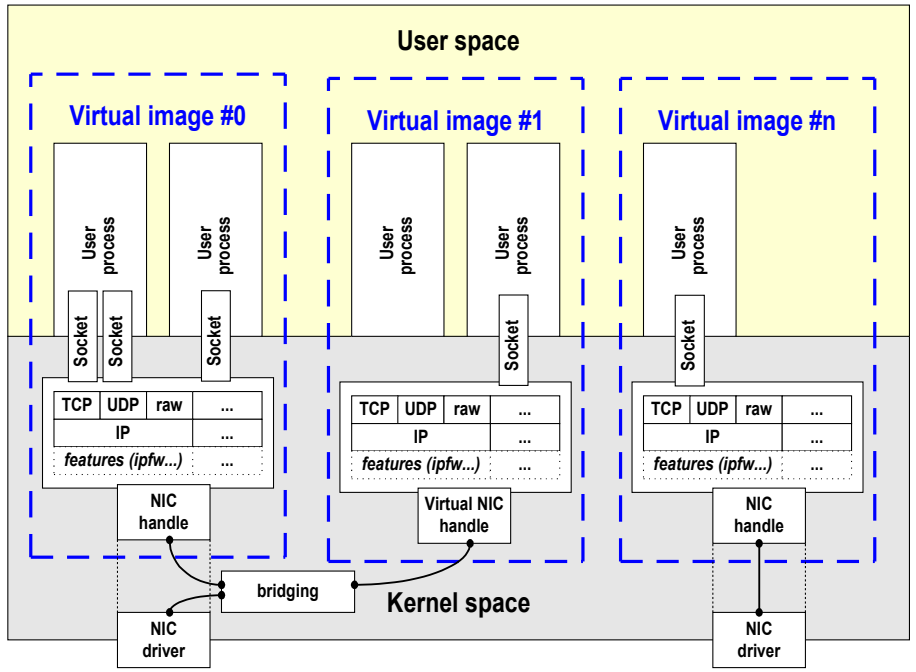
- ng_ether(4)
  - Hooks: upper, lower
  - Diverts / accepts real ethernet traffic to / from another netgraph node
  - Always associated with real ethernet / 802.11 interfaces
  - Automatic naming
- ng_eiface(4)
  - Hooks: ether
  - Virtual interface providing ethernet framing

Netgraph node types

- ng_iface(4)
  - Hooks: inet, inet6, ipx, atalk, ns, atm, natm
  - Virtual interface emiting / accepting raw frames on protocol-specific hooks
- ng_bridge(4) / ng_hub(4)
  - Hooks: link0, link1...
- ng_tee(4)
  - Hooks: left, right, left2right, right2left
  - Snooping / tapping
- Many many more -> man 4 netgraph

Summary: building blocks





- Virtual node -> vimage: private instance of networking state (interfaces, routes, sockets, etc.)
- Netgraph: providing explicit link-layer communication paths between virtual nodes

Exercise 1: booting a virtualized FreeBSD system

- Objectives:
  - Learn basic usage of vimage and ngctl utilities
- Steps:
  - Boot a system with virtualized networking stack
  - Wormup: playing with IMUNES GUI
  - Manually (no GUI!) create three vimages
  - Create four ng_eiface interfaces, connect two * two of them back-to-back
  - Assign the eiface interfaces to vimages, configure them and check the connectivity

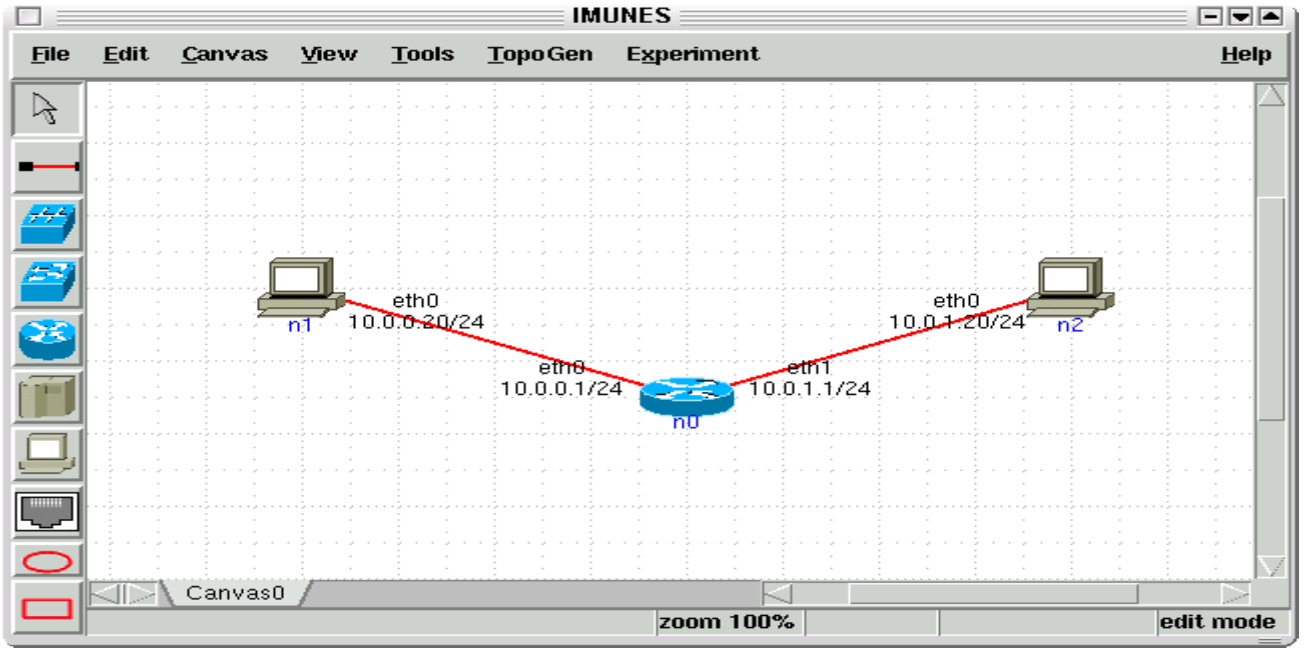
Booting a FreeBSD -CURRENT system

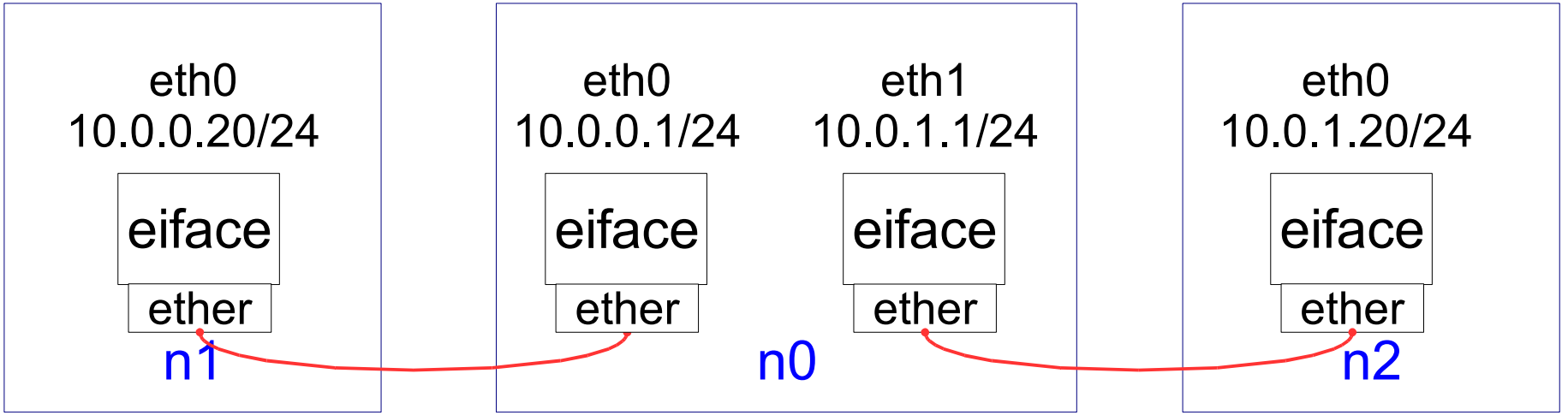
- Adjust booting (startup) sequence in BIOS
  - USB stick must be plugged in when powering up!
  - On recent laptops (ThinkPads, Dells, HPs) pressing F12 after powerup opens a dialog for choosing a boot device
  - With USB 2.0 ports USB operation much faster than LiveCD, can store settings / changes on the stick
- Alternative – VMWare Player
  - WiFi: SSID “virtnet”, no WEP/WPA, IP from DHCP
  - Fetch player + image from http://192.168.200.1/

Booting a FreeBSD -CURRENT system

- Login as root
  - No password!
- “xinit” should start X
  - Edit .xinitrc if you prefer KDE over IceWM
- X does not start?
  - cp /etc/X11/xorg.conf.vesa /etc/X11/xorg.conf
  - black / garbled screen -> CTRL+ALT+Backspace
  - try tweaking the xorg.conf file
  - If nothing helps, try M$ + VMWare player :-)

Exercise 1: target topology





Management interface: vimage

- `vimage` (no arguments)
  - Displays current vimage name
- `vimage -l`
  - Lists all vimages under current position in the hierarchy, including the current one
- `vimage -c name`
  - Creates a new vimage
- `vimage -d name`
  - Deletes a vimage
  - Must have no running processes and open sockets
  - All assigned interfaces returned to parent or destroyed

Management interface: vimage

- `vimage -i name ifname [newifname]`
  - Assigns an interface `ifname` to vimage `name`
  - Interface is automatically renamed to “ethX” when attached in the target vimage. An alternative `newifname` can be specified as an optional argument
  - If “-” is provided instead of target vimage name, the interface is returned to parent vimage, in which case original ifname is automatically restored
- `vimage name command`
  - Executes a `command` in context of vimage `name`
- `vimage name`
  - Spawns an interactive shell in context of the target vimage

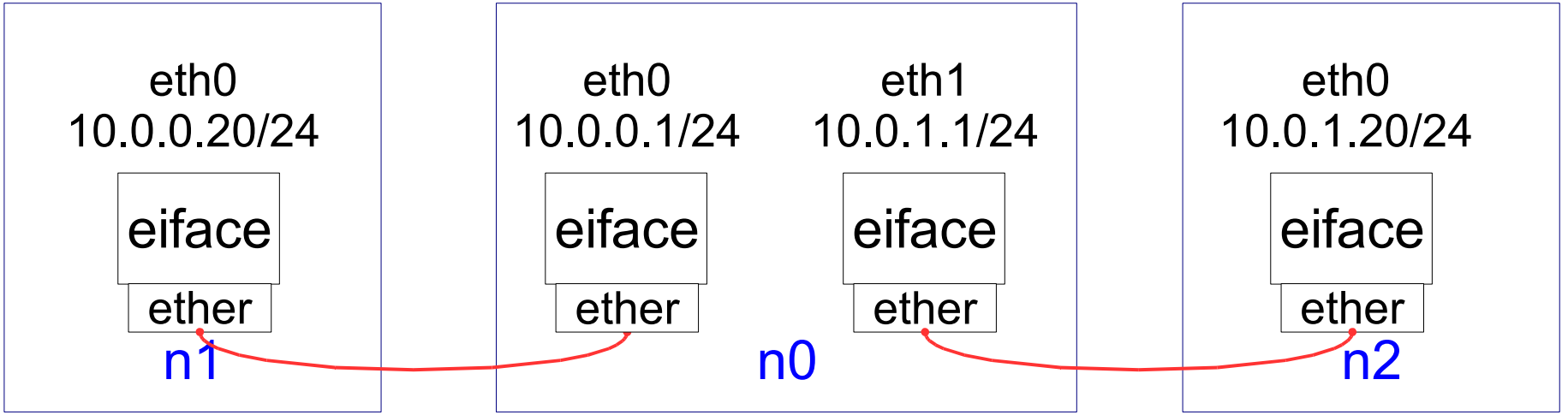
Management interface: ngctl

- `ngctl list`
  - Lists all netgraph nodes in current vimage context
- `ngctl mkpeer [target] type hook1 hook2`
  - Creates a new node of type `type` connected to `target:hook1` with a hook named `hook2`.
  - If `target` is ommited, the new node is connected to the current socket node
- `ngctl name target newname`
  - Sets the symbolic name of node at `target` to `newname`
- Typical addressing format for `target` is `nodename:` or `nodename:hookname`

Management interface: ngctl

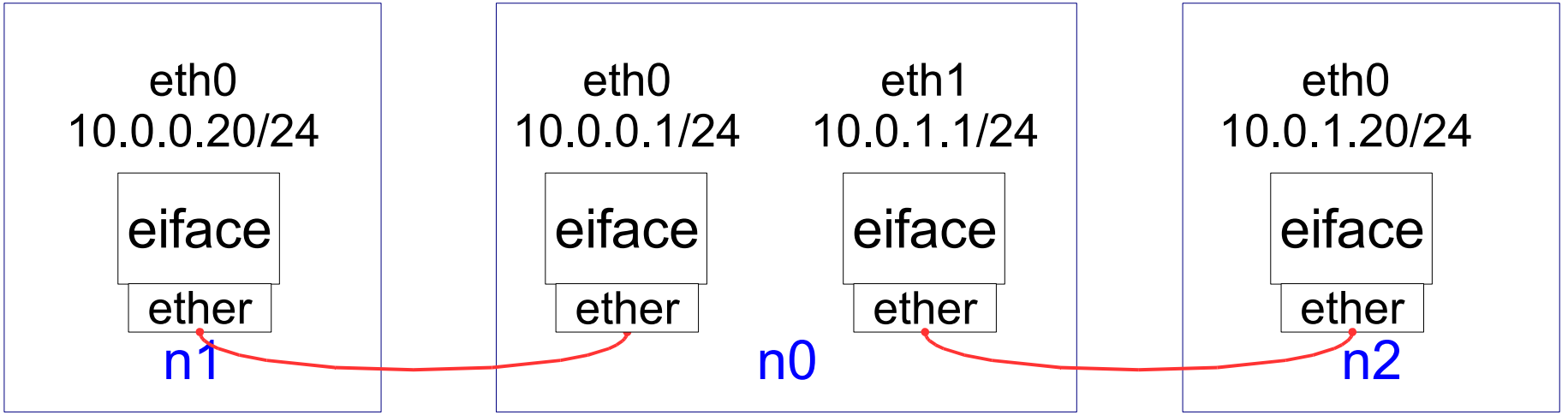
- `ngctl shutdown target`
  - Unconditionally shuts down the target node
- `ngctl connect nodea nodeb hooka hookb`
  - Connects the two nodes
- `ngctl msg target message`
  - Sends a control / management message to a node
  - A response (if received) is displayed
- `ngctl show target`
  - Returns information for each connected hook of the target node

Exercise 1: create virtual nodes



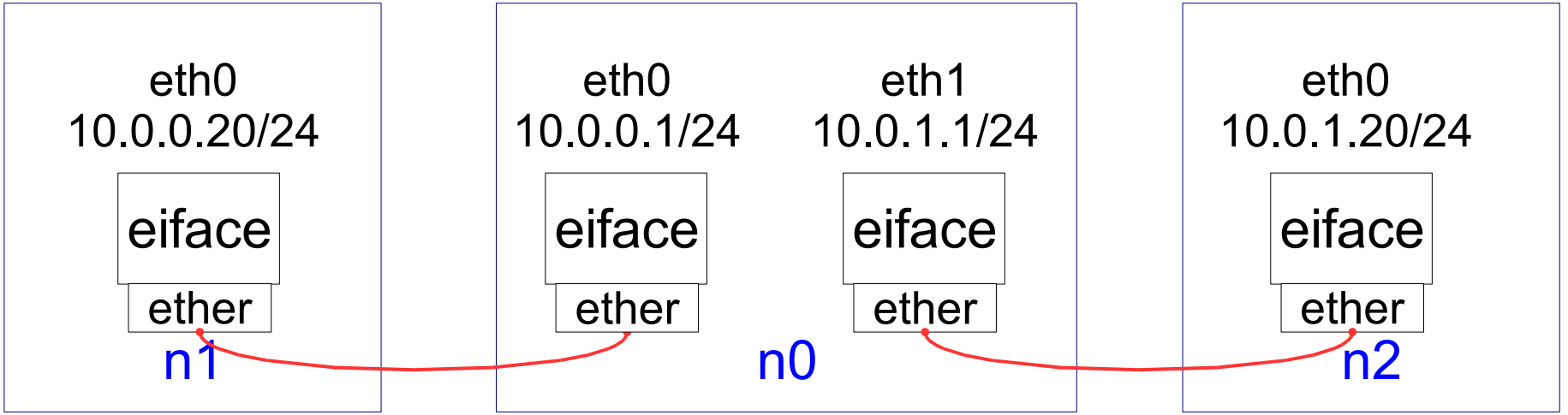

```
tpx32# vimage -l
"default":
    100 processes, load averages: nan, nan, nan
    CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
    CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
    No proc limit
    Sockets (current/max): 139/0
    7 network interfaces
tpx32# vimage -c n0
tpx32# vimage -c n1
tpx32# vimage -c n2
```

Exercise 1: create virtual interfaces



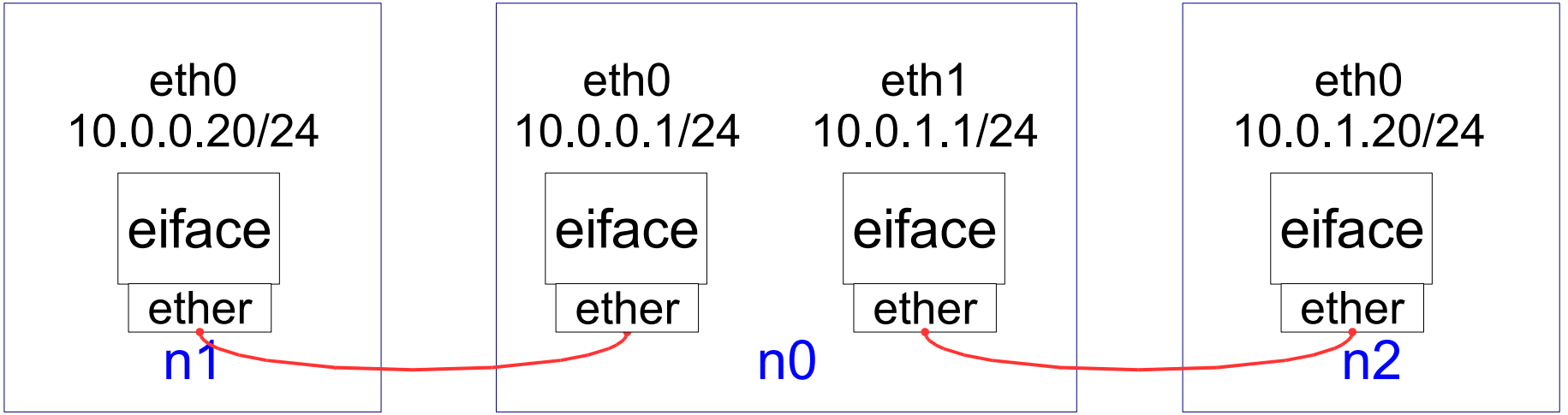

```
tpx32# ngctl l
There are 1 total nodes:
  Name: ngctl5842       Type: socket          ID: 0000001f   Num hooks: 0
tpx32# ngctl mkpeer eiface ether ether
tpx32# ngctl mkpeer eiface ether ether
tpx32# ngctl l
There are 3 total nodes:
  Name: ngctl6195       Type: socket          ID: 00000028   Num hooks: 0
  Name: ngeth1          Type: eiface          ID: 00000023   Num hooks: 0
  Name: ngeth0          Type: eiface          ID: 00000021   Num hooks: 0
tpx32# ifconfig # some output omitted for brevity
ngeth0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:00:00:00:00:00
ngeth1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:00:00:00:00:00
```

Exercise 1: connect and assign virtual interfaces




```
tpx32# ngctl connect ngeth0: ngeth1: ether ether
tpx32# ngctl l
There are 3 total nodes:
  Name: ngctl6509       Type: socket          ID: 0000002a   Num hooks: 0
  Name: ngeth1          Type: eiface          ID: 00000023   Num hooks: 1
  Name: ngeth0          Type: eiface          ID: 00000021   Num hooks: 1
tpx32# vimage -i n0 ngeth0
eth0@n0
tpx32# vimage -i n1 ngeth1
eth0@n1
```

Exercise 1: configure virtual nodes

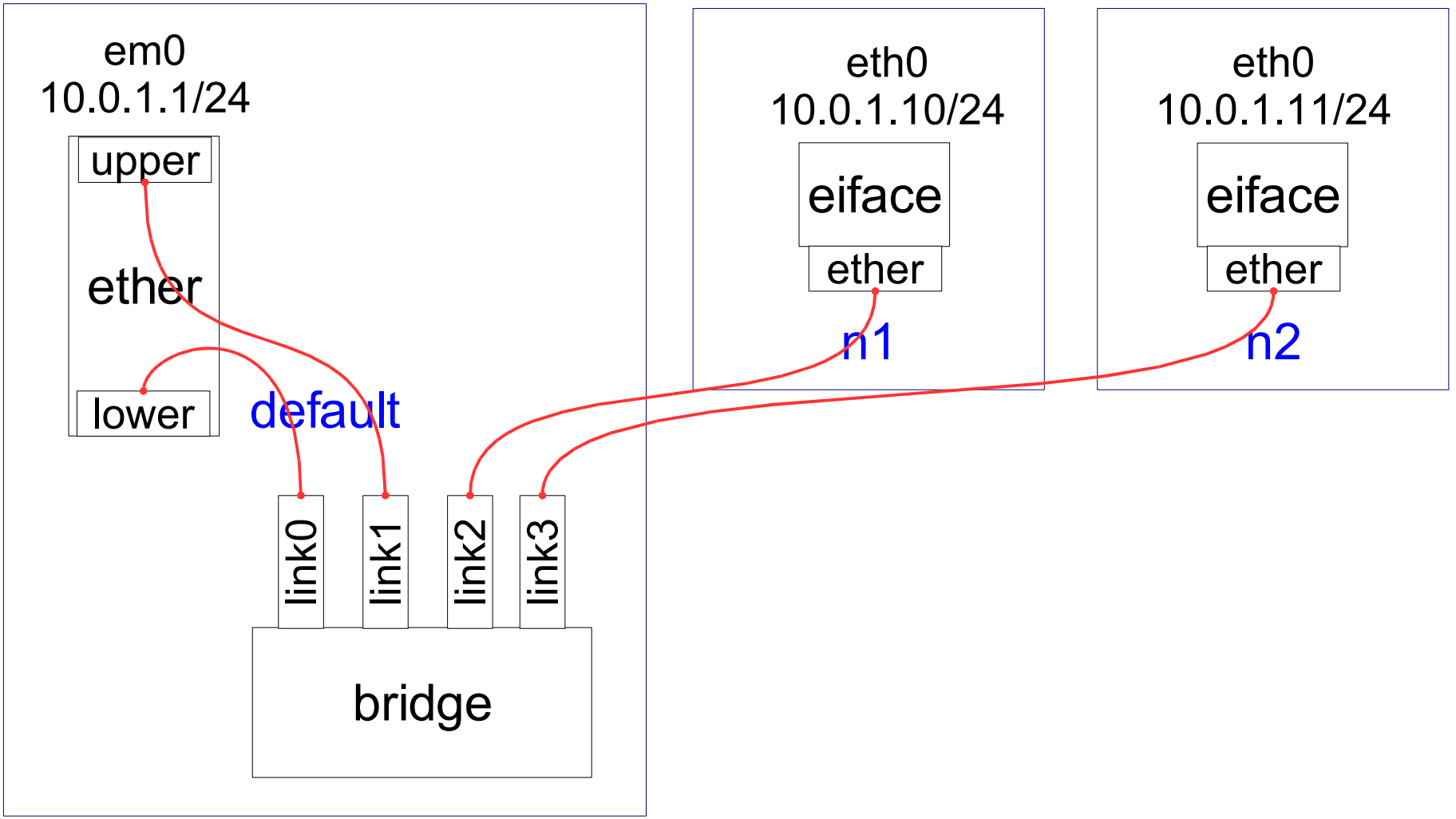



```
tpx32# vimage n1 # only config steps for n1 shown for brevity
Switched to vimage n1
# ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
eth0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:00:00:00:00:00
# ifconfig lo0 localhost
# ifconfig eth0 link 40:a:b:c:d:01
# ifconfig eth0 10.0.0.20/24
# route add default 10.0.0.1
add net default: gateway 10.0.0.1
# hostname n1
...
# ping / traceroute 10.0.1.20
```

Exercise 2: virtual hosting setup

- Objectives:
  - Exposure to more complex vimage / netgraph setup
- Steps:
  - Clean up the old config (vimages, netgraph nodes)
  - Manually (no GUI!) create two vimages
  - Create two ng_eiface interfaces and a ng_bridge instance and connect them
  - Connect a physical ethernet interface to bridge
  - Assign the eiface interfaces to vimages, configure; verify connectivity; play with ipfw firewall

Exercise 2: target topology



Exercise 2: clean up old config

```
tpx32# vimage -d n0
tpx32# vimage -d n1
tpx32# vimage -l
"default":
    102 processes, load averages: nan, nan, nan
    CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
    CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
    No proc limit
    Sockets (current/max): 135/0
    4 network interfaces
tpx32# ngctl shutdown ngeth0:
tpx32# ngctl shutdown ngeth1:
tpx32# ngctl l
There are 1 total nodes:
  Name: ngctl8474       Type: socket          ID: 00000033   Num hooks: 0
tpx32#
```

Exercise 2: create vimages and virtual interfaces

```
tpx32# vimage -c n1
tpx32# vimage -c n2
tpx32# ngctl mkpeer eiface ether ether
tpx32# ngctl mkpeer eiface ether ether
tpx32# ngctl l
There are 3 total nodes:
  Name: ngctl8856       Type: socket          ID: 00000039   Num hooks: 0
  Name: ngeth1          Type: eiface          ID: 00000038   Num hooks: 0
  Name: ngeth0          Type: eiface          ID: 00000036   Num hooks: 0
tpx32# kldload ng_ether
tpx32# ngctl l
There are 5 total nodes:
  Name: ngctl8897       Type: socket          ID: 0000003c   Num hooks: 0
  Name: iwi0            Type: ether           ID: 0000003b   Num hooks: 0
  Name: em0             Type: ether           ID: 0000003a   Num hooks: 0
  Name: ngeth1          Type: eiface          ID: 00000038   Num hooks: 0
  Name: ngeth0          Type: eiface          ID: 00000036   Num hooks: 0
```

Exercise 2: create a bridge and connect interfaces

```
tpx32# ngctl mkpeer em0: bridge lower link0
tpx32# ngctl l
There are 6 total nodes:
  Name: ngctl9290       Type: socket          ID: 00000040   Num hooks: 0
  Name: <unnamed>       Type: bridge          ID: 0000003f   Num hooks: 1
  Name: iwi0            Type: ether           ID: 0000003b   Num hooks: 0
  Name: em0             Type: ether           ID: 0000003a   Num hooks: 1
  Name: ngeth1          Type: eiface          ID: 00000038   Num hooks: 0
  Name: ngeth0          Type: eiface          ID: 00000036   Num hooks: 0
tpx32# ngctl name em0:lower bridge0
tpx32# ngctl connect em0: bridge0: upper link1
tpx32# ngctl connect ngeth0: bridge0: ether link2
tpx32# ngctl connect ngeth1: bridge0: ether link3
tpx32# ngctl l
There are 6 total nodes:
  Name: ngctl9349       Type: socket          ID: 00000045   Num hooks: 0
  Name: bridge0         Type: bridge          ID: 0000003f   Num hooks: 4
  Name: iwi0            Type: ether           ID: 0000003b   Num hooks: 0
  Name: em0             Type: ether           ID: 0000003a   Num hooks: 2
  Name: ngeth1          Type: eiface          ID: 00000038   Num hooks: 1
  Name: ngeth0          Type: eiface          ID: 00000036   Num hooks: 1
```

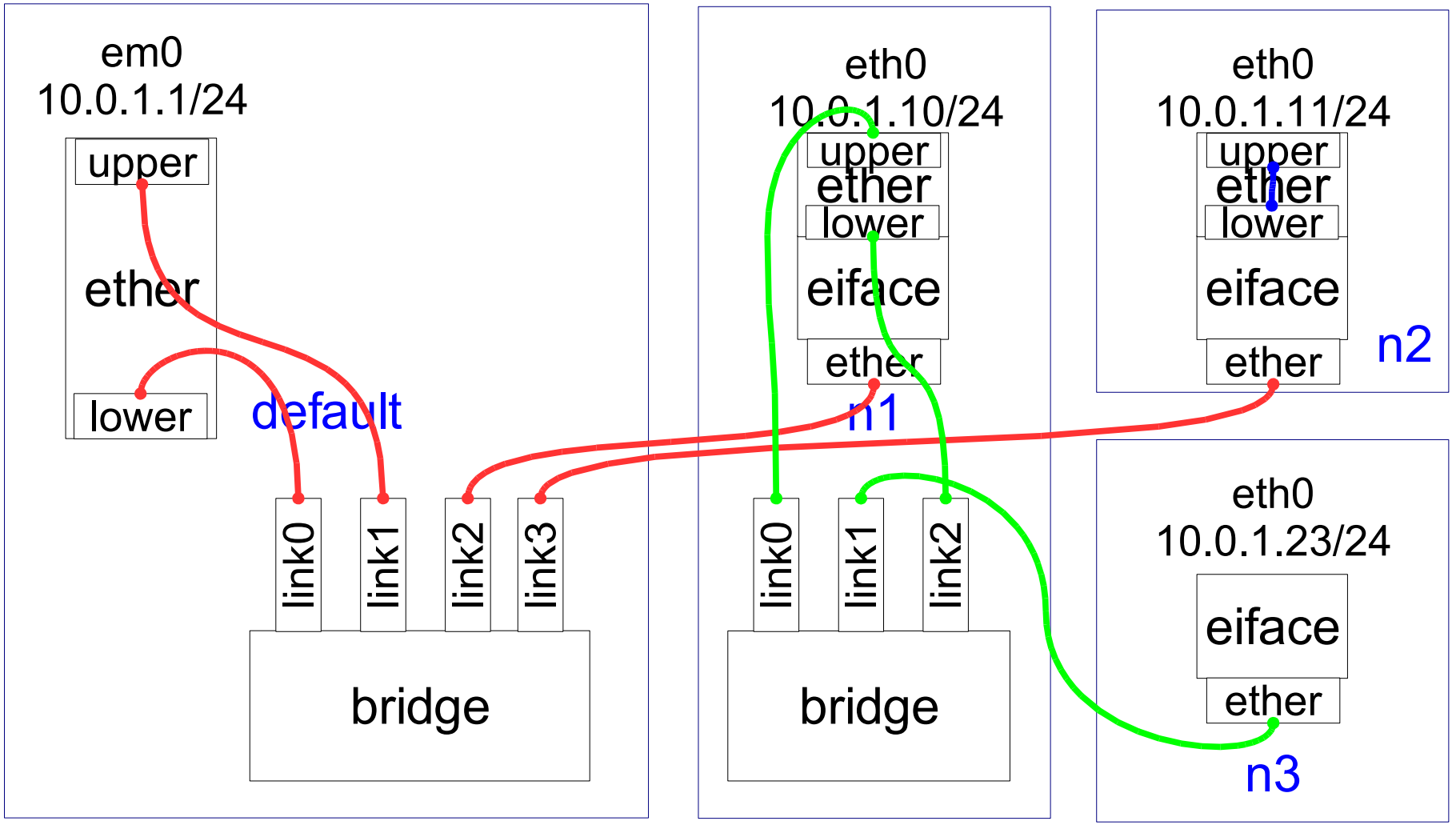
Exercise 2: assign and configure interfaces

```
tpx32# vimage n1 ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
tpx32# vimage -i n1 ngeth0 e0
e0@n1
tpx32# vimage n1 ifconfig lo0 localhost
tpx32# vimage n1 ifconfig e0 link 40:a:b:c:d:01
tpx32# vimage n1 ifconfig e0 10.0.1.10/24
tpx32# vimage n1 ping 10.0.1.11
PING 10.0.1.11 (10.0.1.11): 56 data bytes
^C
--- 10.0.1.11 ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
tpx32# kldload ipfw
tpx32# vimage n1 ping 10.0.1.11
PING 10.0.1.11 (10.0.1.11): 56 data bytes
ping: sendto: Permission denied
^C
--- 10.0.1.11 ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
tpx32# vimage n1 ipfw list
65535 deny ip from any to any
tpx32# vimage n1 ipfw add 100 accept ip from any to any
00100 allow ip from any to any
```

Exercise 2a: nested virtual hosting setup

- Objectives:
  - Exposure to more complex vimage / netgraph setup
- Steps:
  - In vimage n1 create a ng_eiface interface and a ng_bridge instance and connect them
  - In vimage n1 connect parent-inherited ethernet interface to bridge
  - From vimage n1 assign the eiface interface to vimage n3, configure; verify connectivity

Exercise 2a: target topology



Exercise 2a: create a new vimage from n1

```
tpx32# vimage n1 ngctl l
There are 2 total nodes:
  Name: ngctl1047       Type: socket          ID: 00000003   Num hooks: 0
  Name: eth0            Type: ether           ID: 00000002   Num hooks: 0
tpx32# vimage n1
Switched to vimage n1
n1# vimage -c n3
n1# vimage -l
"n1":
    1 processes, load averages: nan, nan, nan
    CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
    CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
    No proc limit
    Sockets (current/max): 1/0
    2 network interfaces, parent vimage: "default"
"n3":
    0 processes, load averages: nan, nan, nan
    CPU usage: (0.00% user, 0.00% nice, 0.00% sys, 0.00% intr)
    CPU limits: min 0.00%, max 100.00%, weight 0, no intr limit
    No proc limit
    Sockets (current/max): 0/0
    1 network interfaces, parent vimage: "n1"
```

Exercise 2a: connect interfaces to a new bridge

```
tpx32# vimage n1
Switched to vimage n1
n1# vimage n3 hostname n3
n1# ngctl mkpeer eiface ether ether
n1# ngctl mkpeer ngeth0: bridge ether link0
n1# ngctl name ngeth0:ether bridge_n1
n1# ngctl connect eth0: bridge_n1: lower link1
n1# ngctl connect eth0: bridge_n1: upper link2
n1# ngctl l
There are 4 total nodes:
  Name: ngctl1472       Type: socket          ID: 0000000d   Num hooks: 0
  Name: bridge_n1       Type: bridge          ID: 00000008   Num hooks: 3
  Name: ngeth0          Type: eiface          ID: 00000005   Num hooks: 1
  Name: eth0            Type: ether           ID: 00000002   Num hooks: 2
n1# vimage -i n3 eth0
eth0@n3
```

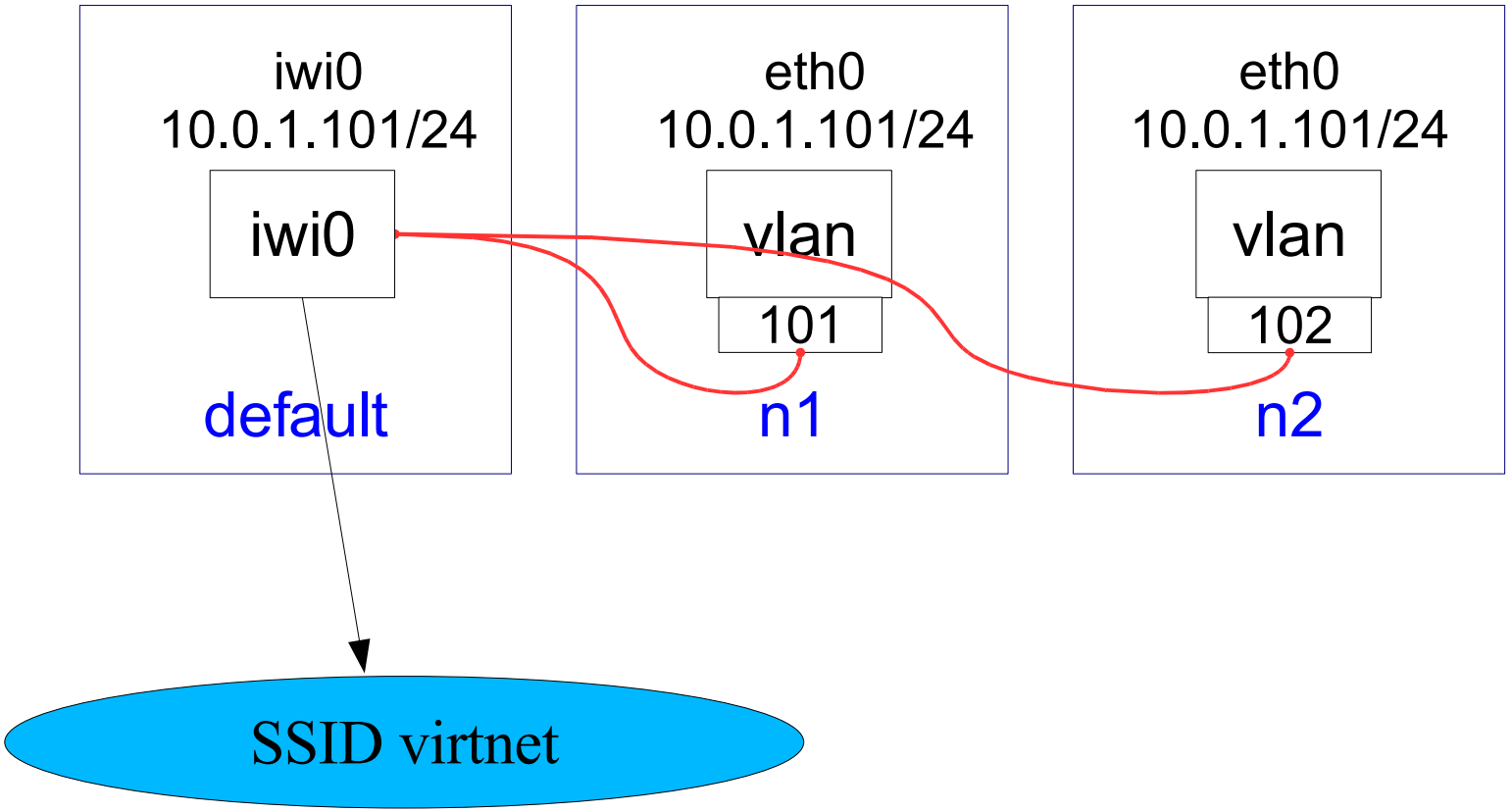
Exercise 2a: configure new vimage

```
n1# vimage n3
Switched to vimage n3
n3# ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
eth0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:00:00:00:00:00
n3# ifconfig lo0 localhost
n3# ifconfig eth0 link 40:a:b:c:d:03
n3# ifconfig eth0 10.0.1.23/24
n3# ping 10.0.1.1
PING 10.0.1.1 (10.0.1.1): 56 data bytes
^C
```

Exercise 3: access to L2 VPNs

- Objectives:
  - Multiplexing access to networks with overlapping addressing scheme over a single external link
- Steps:
  - Clean up the old config (vimages, netgraph nodes)
  - Create two vimages
  - Create two VLAN interfaces using different VLAN tags associated with a physical wifi interface
  - Assign VLAN interfaces to vimages; configure; attempt / check connectivity with others

Exercise 3: target topology



Exercise 3: create vimages and vlan interfaces

```
tpx32# vimage -c n1
tpx32# vimage -c n2
tpx32# kldload if_vlan
tpx32# ifconfig iwi0.101 create
tpx32# ifconfig iwi0.102 create
tpx32# ifconfig
iwi0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:16:6f:37:f8:36
        # output omitted for brevity
iwi0.101: flags=8842<BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1496
        ether 00:16:6f:37:f8:36
        media: IEEE 802.11 Wireless Ethernet autoselect
        status: associated
        vlan: 101 parent interface: iwi0
iwi0.102: flags=8842<BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1496
        ether 00:16:6f:37:f8:36
        media: IEEE 802.11 Wireless Ethernet autoselect
        status: associated
        vlan: 102 parent interface: iwi0
tpx32# vimage -i n1 iwi0.101
eth0@n1
tpx32# vimage -i n2 iwi0.102
eth0@n2
```

*Exercise 3: **identical** IP address on all ifcs*

```
tpx32# ifconfig iwi0 10.0.1.101.24/24
tpx32# vimage n1 ifconfig eth0 10.0.1.101/24
tpx32# vimage n2 ifconfig eth0 10.0.1.101/24
tpx32# vimage n1 ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
eth0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:16:6f:37:f8:36
        inet 10.0.1.101 netmask 0xffffff00 broadcast 10.0.1.255
        inet6 fe80::d41d:8cd9:8f00:b204%eth0 prefixlen 64 scopeid 0x2
        media: IEEE 802.11 Wireless Ethernet autoselect (DS/2Mbps)
        status: associated
tpx32# vimage n2 ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
eth0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> metric 0 mtu 1500
        ether 00:16:6f:37:f8:36
        inet 10.0.1.101 netmask 0xffffff00 broadcast 10.0.1.255
        inet6 fe80::d41d:8cd9:8f00:b204%eth0 prefixlen 64 scopeid 0x2
        media: IEEE 802.11 Wireless Ethernet autoselect (DS/2Mbps)
        status: associated
tpx32# ping 10.0.1.102
tpx32# vimage n2 ping 10.0.1.102
```

Exercise 3: check / attempt connectivity with others

```
tpx32# ping 10.0.1.102
PING 10.0.1.102 (10.0.1.102): 56 data bytes
^C
--- 10.0.1.102 ping statistics ---
1 packets transmitted, 0 packets received, 100.0% packet loss
tpx32# vimage n2 ping 10.0.1.102
PING 10.0.1.102 (10.0.1.102): 56 data bytes
64 bytes from 10.0.1.102: icmp_seq=0 ttl=64 time=3.424 ms
64 bytes from 10.0.1.102: icmp_seq=1 ttl=64 time=3.131 ms
^C
--- 10.0.1.102 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 3.131/3.277/3.424/0.147 ms
tpx32# vimage n2 ifconfig eth0 down
tpx32#
```

Shuffle addresses and interfaces up/down

Look at arp caches etc.

Implementation in FreeBSD 7.0-CURRENT

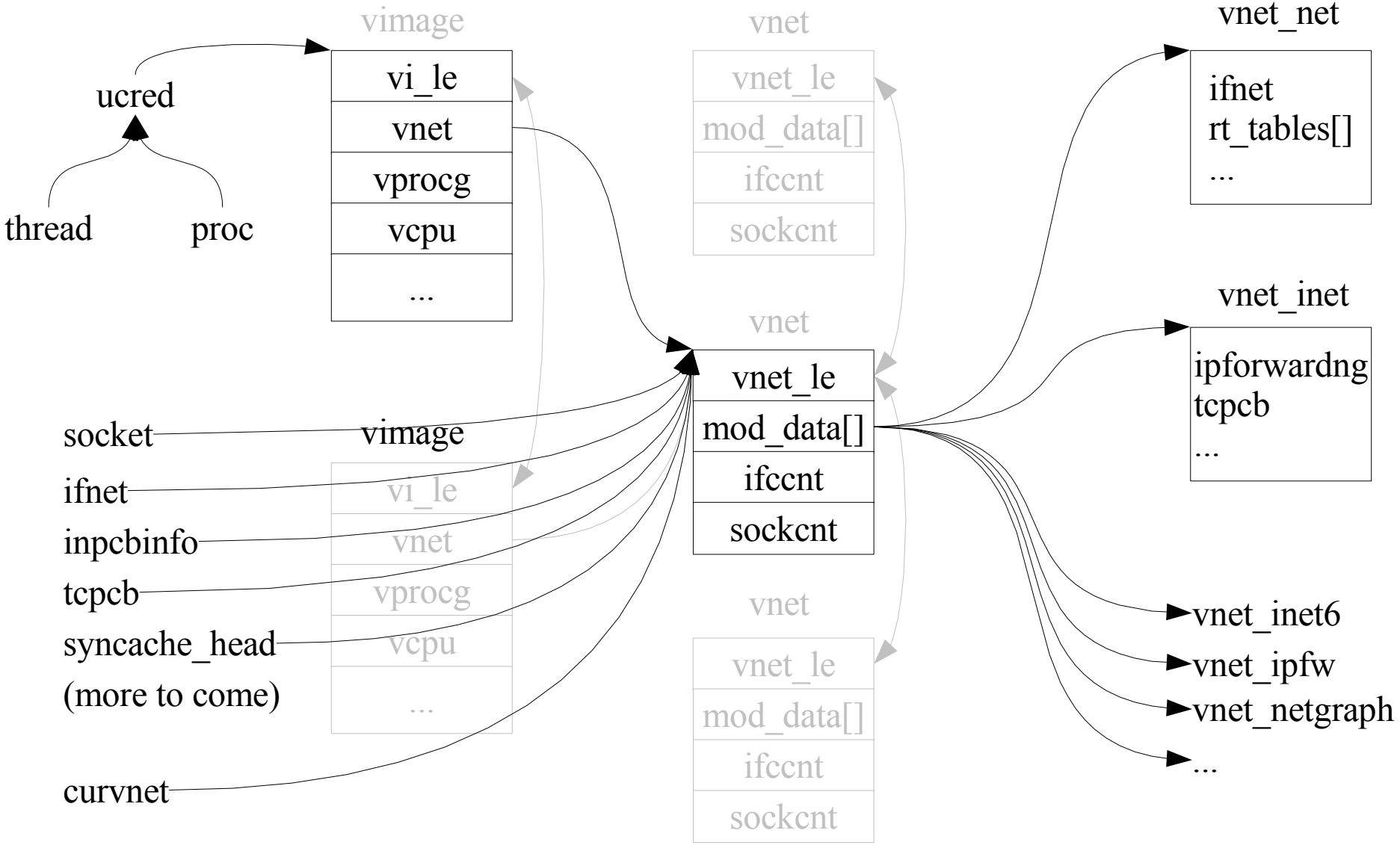
Implementation concepts: long time ago...

- Patches against FreeBSD 4.7 .. 4.11 kernels
  - Unsupported platform today
- `struct vnet`
  - One huge structure / container; each network stack instance operates on its private copy
  - Contains ifnet lists, IPv4 / IPv6 / firewall state etc.
- Unconditional changes to original kernel code
  - Each networking function got an additional argument: `struct vnet *`

(Re)implementation: 7.0

- Goals:
- Conditional compilation
- Better support for kernel loadable modules
- Scope of changes is huge: reduce code churn
- SMP from day 0
- Otherwise, no chances for including the changes into main FreeBSD tree

Replicate global networking state: how?



vnet modules: registration / deregistration

```
static struct vnet_symmap vnet_net_symmap[] = {
        VNET_SYMMAP(net, ifnet),
        VNET_SYMMAP(net, rt_tables),
        ...
        VNET_SYMMAP_END
};

VNET_MOD_DECLARE(NET, net, vnet_net_iattach, vnet_net_idetach,
    NONE, vnet_net_symmap)

/*
VNET_MOD_DECLARE(IPFW, ipfw, vnet_ipfw_iattach, vnet_ipfw_idetach,
INET, NULL)
 */

if_init(void *dummy __unused)
{
#ifdef VIMAGE
        vnet_mod_register(&vnet_net_modinfo);
#else
        vnet_net_iattach();
#endif
        ...
```

Conditional compilation: `option VIMAGE`

- Dereference virtualized symbols: how?
  - Use macros for this. Example:
    - `if_addrhead` becomes **`V`**`_if_addrhead`
  - Standard kernel:
    - `V_if_addrhead` expands back to `if_addrhead`
  - Virtualized kernel:
    - `V_if_addrhead` expands to `vnet_net->_if_addrhead`
  - Sysctl and kldsym interfaces extended to support access to virtualized symbols

Reducing code churn

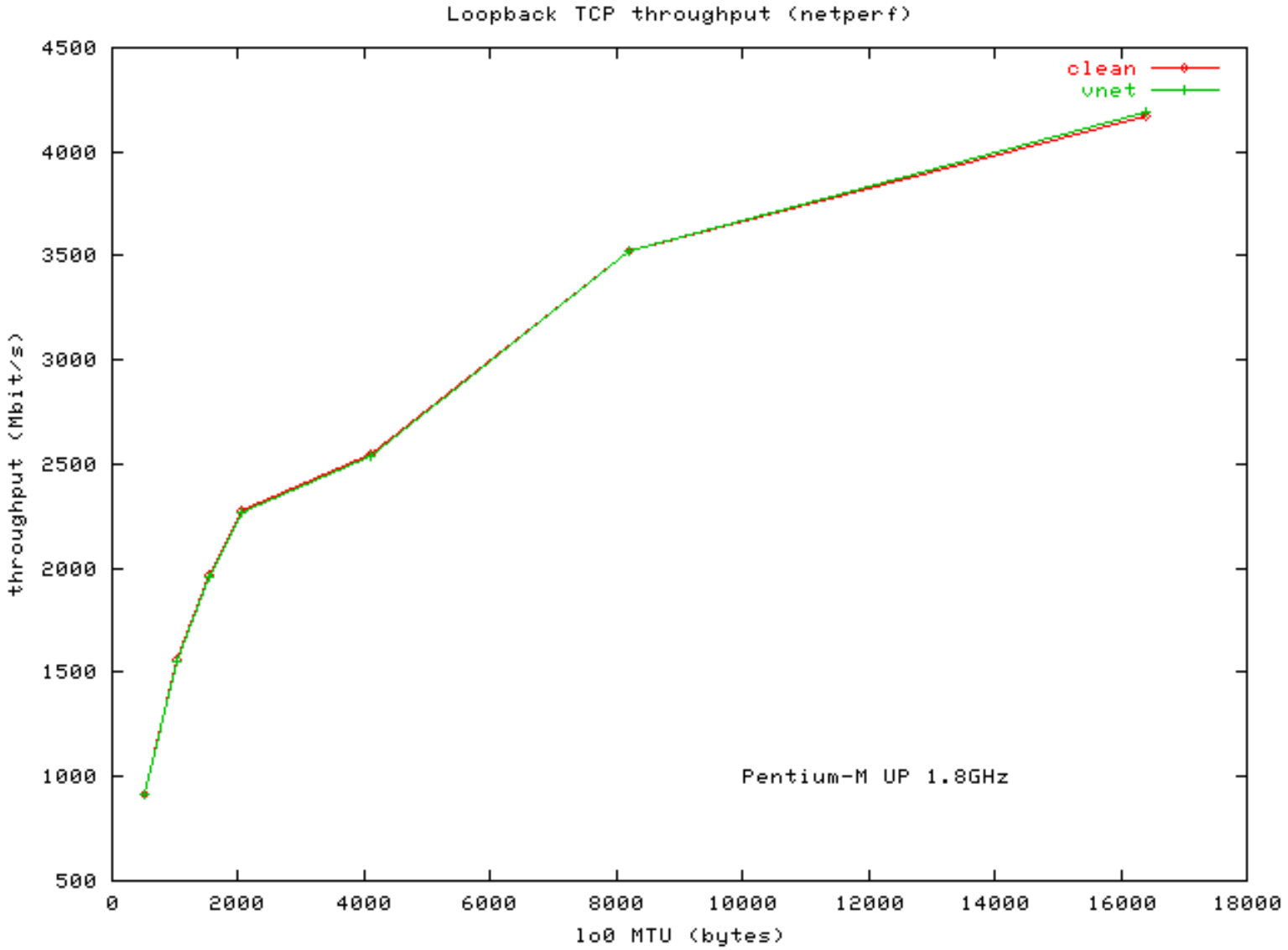
- Implicitly pass the `vnet` context to operate on:
  - Thread-local `curvnet` variable

```
void if_attach(struct ifnet *ifp)
{
        INIT_VNET_NET(curvnet);
        ...
}
```

`INIT_VNET_NET(x)` (x is a `struct vnet *`) expands to

```
struct vnet_net *vnet_net = x->mod_data[VNET_MOD_NET];
```

Performance: loopback TCP throughput



Generalizing OS-level virtualization

- Management concepts / API
  - Top-level resource container `struct vimage`
  - Contains freely combinable subsystem-specific state
    - `vnet, vcpu, vprocg, vfs...`
  - Single process with sockets in multiple stacks
    - Extend socket interface -> multi-table routing daemons
  - Hierarchy of vimages – follow UNIX process model?
  - Permissions, restrictions, inheritance...
  - How to best integrate those new concepts / features with the rest of the system?

To conclude...

- Do we need all this?
  - the community has to provide that answer.
- If yes, what's next to virtualize?
  - CPU time (scheduler)
  - Filesystems (ZFS?) / disk I/O bandwidth
  - Memory
  - ...
- We need a generalized OS-level virtualization model

http://imunes.tel.fer.hr/virtnet/