Netgraph in 5 and beyond

What Netgraph is, what changed, and why.

Netgraph's goals

- Simple modules with minimal overhead code. (don't worry about the infrastructure, just worry about what you are trying to do).
- Able to prototype a node type in userland. This means the userland interface must be not too dissimilar to the internal node interface.
- Allow arbitrary orderring of nodes.
- Allow a sysad to configure a node without having to install a utility for every node type.
- Able to connect many nodes to one node.

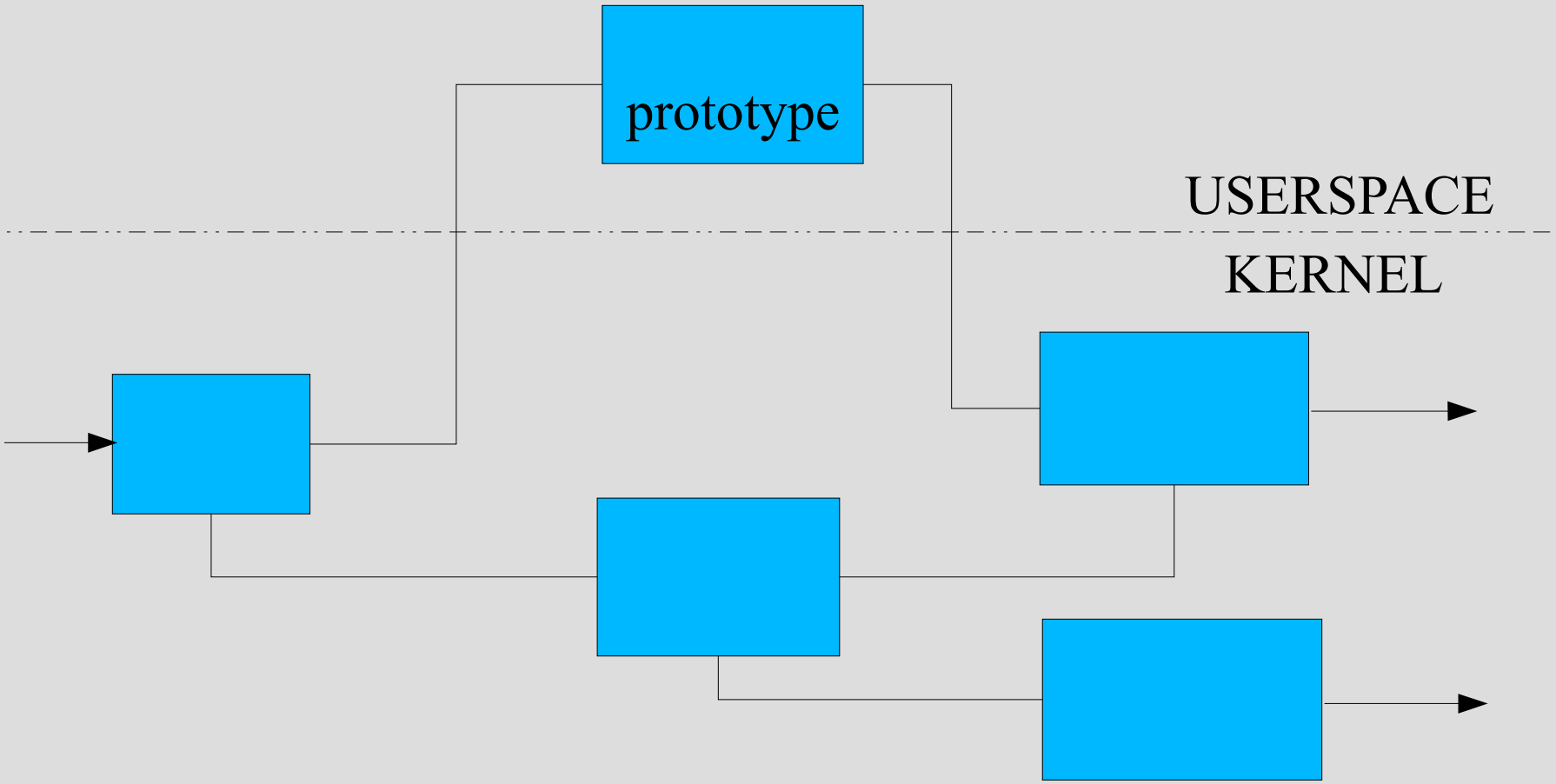
Netgraph's goals (cont.)

- Data in transit to have minimal delays at the expense of more expensive configuration operations.
- Ability to pair metadata with inline data. (e.g. Packet priority)
- Able to be connected to other system components as a source or sink of data.
- In SMP systems, data travelling in opposite directions should be able to pass without causing problems (at least, none from the framework). (New)

Netgraph's goals (cont.)

- No built-in concept of up or down.
- Ability to address any node for configuration.
- Ability for a node to generate configuration requests for another node.
- Ability to associate a node with a meaningful name.
- Nodes types to be loadable at run time.
- Netgraph framework to be loadable at run time.
- Data moving between nodes could be immediately used or queued. (node decides).

A sample configuration. (Appears graph like, hmmmm).



Original Design.

- Based on the original BSD scheme of spl() with nodes generally running at netisr() priority level with code associated with hardware generally running at splimp() level.
- Implied that we needed to queue data from edge nodes and internal nodes could pass data directly (probably).
- Relied on there only being one processor in the code at one time.
- Nodes could force queueing if the had a reason to do so.

Other influences.

- I always liked the modularity of sysV STREAMS.
- I didn't like some of the limitations as to what you could do with them, e.g. It was heavily bases on a protocol “stack” idea with inbuilt ideas about “up” and “down”.
- I didn't like having to disassemble the stack to configure nodes.
- I didn't like the requirement to queue items between elements.

Other influences.

- The BSD modules had flexible interfaces but they had been warped so much byu special cases that they were no longer general enough.
- The BSD interfaces were different at each level.
- Needed a new idea. In particular I wanted to be able to configure a new node without having to load a new userland utility to do it.

The basic idea.

- Give the nodes the ability to have “hooks”.
- Give the hooks the ability to have names.
- Give the nodes the ability to recognise special names. This allows simple configuration just by selcting the right names for the connections.
- Make the connections between hooks symetrical.
- Hooks only exist whan connected to another hook.
- Hooks can connect to only one other hook.

The basic idea.

- A node may have a LOT of hooks.
- A node can have its own way of looking up hooks, and can associate private data with each hook.
- ***In 5.x(+) a node may associate an over-ride data handler method with a hook*.**
- A hook is always either connected to another hook or it is destroyed. (* “dead” node).
- A second data type called a “control message” canbe routed between nodes.

The basic idea.

- Control messages are routed to an endpoint.
- An endpoint may be specified absolutly or relatively.
  - Similar to 'bang” routing in uucp, e.g. Specify a sequence of hooks ro follow to get to the destination.
  - A node could send a message to a neighbour without knowing its name but only how to get there.
- Control messages can be sent from Userland. Allows user utilities to configure nodes.

The basic idea.

- Each node type can specify message formats.
- Standard message headers allow extensible formats.
- Brilliant work by archie@ allows ascii control messages, giving the ability to configure any node with a single simple utility (ngctl).
  - Could probably do with work after 9 years but still very unique.
- Each type has its own 'cookie' that makes its own API uniquely distinguishable.

The basic idea.

- Nodes are permitted to implement a message ABI defined in the include files of another type. It just needs to check the cookie.
- Generic operations are implemented as messages with a “generic” cookie, that are handled by the framework immediately before they would be handed to the node. Sometimes this results in methods for the node being called instead. e.g. Shutdown.

Growth.

- Many node type now exist:
- PPP, Frame relay, tee, ether, socket, ksocket, pppoe, bpf-filter*, bridging, ipfw-tie-in, hdlc, tty-protocol, an entire bluetooth stack, an entire atm stack, cisco netflow accounting hooks, etc. etc.
- See http://www.freebsd.org/cgi/cvsweb.cgi/src/sys/netgraph/

* filter using a bpf/tcpdump compiled 'filter'.

Why change in 5.x (+)?

- Basically, SMP.
- Any node may call any node at the same time as another cpu is also in that node.
- Blocking all of netgraph for the time it takes to process a packet would be to clunky.
- splx/splimp/splnet going right away.

Aims for the changes

- Clean up stuff I was embarrased over.
- Add per node locking.
- Make the locking as cheap as possible for common operations.
- Integrate locking and queueing.
- Provide a way for timeouts and external callers to participate in locking.
- Switch to using system provoded metadata facilities (mbuf tags).

Locking

- Divide locking into reader and writer locks.
  - Data traversing a node is a reader if it doesn't change the state of the node. Many readers may be in a node at one time.
  - Control messages are by default writers. They are expected to do something with the state of the node. (even if just look at it) and are expected to want exclusive access.
  - Existence of a writer forces incoming readers to be queued.
  - Existence of one or more readers or writers forces an incoming writer to be queued.
  - Queued items are always serviced FIFO.

Locking (cont)

- A node may declare that all incoming data should obtain writer locks if it knows that it will change node state with plain data.
- A node may set up a single hook on which data enterring must obtain a writer lock.
- A node may set the force-writer bit on a PEER node. This is useful for devices who want to ensure that the outgoing data is quickly queued and that ongoing processing will be done at another time.

Locking (cont)

- Failure to get a lock will result in the operation being queued, therefore ALL operations must be queueable.
- Obvious cases:
  - Data can be queued instead of being processed.
  - Controll messages can beb queue
- Less obvious cases:
  - Queuing function calls
    - Requires 'void' type.. mno return value
  - Timeout functions queue their actions when they 'expire'

Reference counting

- All queued operations hold references on the node and if they reference a hook, they hold a reference on that too.
  - The netgraph framework tries to be obsesive about reference counting. Almost any pointer to a node or hook is reference counted. Even internal references between nodes and hooks.
- As a result a queued ore deferred operation should not be able to run and find that its target node has gone away. It may be labled Invalid, but it will still exist until the last reference has gone away.

MACROS

- MACROs are extensively used in netgraph.
- They are generally defined in netgraph.h and are used to hide extensive debugging code.
- Also used to provide compatibility (or at least assist) between 4.x netgraph and 5.x netgraph.

Ok, details

- Netgraph node methods are largly unchanged. Exceptions are the methods for receiving data and control messages.
- There used to be 2 data reception methods
  - One for directly applied data
  - One for data that could be queued.
- There is now just one. The queuing functionality is now achieved using a flag on either the node or the hook.

Ok, details

- Methods for data and message reception have simplified arguments.
  - The “Item” structure that was occasionally used to queue items has been made more ubiquitous and is used throughout. The “Item” is preloaded with pointers to
    - Data (including metadata)
    - Control message.
    - Function to call and arguments.
    - Pointers to node and hook (if relevent).
    - Return address if a message.
  - Thus any of these can be queued at any time because the “Item” is a queue item.
  - Holds a lot of what were arguments.

dunno