Modernizing the BSD Networking Stack

Dennis Ferguson

About Me

- I've been a Unix (v7) user since about 1980
- I learned to love networking by reading, using and modifying the BSD network stack, starting with 4.2BSD (I think a beta version) in 1983
- I have developed software for four BSD-based host-in-router projects
  - the original CA*net routers, ca. 1990
  - the T3 NSFnet routers (AIX host), ca. 1992
  - Ipsilon Networks routers for a while, ca. 1995
  - Juniper Networks routers, since 1996
- I've acquired some opinions in the process

About This Talk

- I started off thinking I could may write a polemic about the BSD network stack and how many things I had to do to make it useful as a host-in-a-router, but I found I couldn't. The BSD network stack is good, it is just a bit old and its roots in a simpler time are showing
- Never-the-less I did have to do a fair amount of work to turn this into something that could be used by a good (for its time) router. While the implementation was a quick hack the basic organization turned out to be good and has lasted far longer than I'd imagined it might
- What I want to talk about are some basic organizational bits which were changed in the stack which turned out to have significant utility. This mostly deals with interface representation and route maintenance; changing these two bits made all other issues easier
- I also believe the best result for multicore scaling is obtained, in both a router and a host, if the basic stateless packet forwarding can be done without locks at all. This requires that packet processing access global data through a limited number of structures which can be modified while being read. We didn't start off with this in mind but it ended up what we did do was the right shape. I won't talk much about this, but you might keep it in mind that this is one of the places this is going.

Hosts versus Routers

- Hosts deal in packets which are somehow addressed to or originated by, or for, applications running locally
- Routers additionally deal with packets which are not of interest to local applications, but are instead forwarded directly from one interface to another.
- You can route packets through the same kernel that supports local applications (BSD has always supported this) but at some scale it makes sense to move if→if forwarding to dedicated hardware. All but the first router I worked on did the latter.
- Even when forwarding is done elsewhere the host-in-the-router remains to support local applications.

Hosts versus Hosts-in-Routers

- Apart from packet forwarding, networking loads for hosts-in-routers may differ from that for hosts
  - Routers (and hence hosts-in-routers) typically have more interfaces, more direct neighbours and more local addresses.
  - Routers typically need to maintain a larger number of routes to make correct routing choices for locally-originated packets.
- These are solely differences of scale. A good host-in-a-router will also be a good, scalable host.

BSD as a Host-in-a-Router

- It is difficult to use an unmodified BSD network stack for a modern host-in-a-router
- The fundamental difficulties with this seem to have three root causes:
  - The representation of interfaces in a BSD kernel
  - The representation of routes and next hops, and the procedures for determining routing
  - The embedded assumptions of limited scale
- I would like to explore some aspects of these, describing what the issues are and what we ended up doing to improve the situation while reorganizing for better MP scaling
- As noted previously, I believe work to provide better networking support for a host-in-a-router also provides better host networking

Network Interfaces: The Hardware View

- Most people are pretty clear about the hardware view of network interface hardware:
  - It has a connector (or antenna) to attach it to the network physically
  - It has a physical/logical layer state (up/down/…)
  - It may require configuration of physical layer parameters, like media selection or clocking
  - It has an output queue to buffer mismatches in capacity and demand
  - It has an L2 encapsulation (or several alternatives) which typically may include L2 station addresses and packet protocol identifiers

Network Interfaces: The Protocol View

- IP (including IPv6) sends packets to and receives packets from interfaces, but what it defines as an “interface” is not the same as the hardware view
- For IP an “interface” is now almost always defined by multicast scope. The neighbours on a single interface are those that can receive a single multicast/broadcast addressed packet sent out the interface. Neighbours that can’t receive that packet are on another interface.
- This view of “interface” is imposed or encouraged by:
  - The IP multicast service
  - IP routing protocols, generally
  - ARP and ND
- This is a fairly new problem. When the BSD network stack was originally implemented none of those protocols were used (not even ARP). See RFC796 and RFC895 for the types of networks for which the original stack was designed.
- Note that while NBMA networks still find support in IP routing protocols, in practice more recent NBMA networks (ATM, Frame Relay) are (or, were) generally treated as multiple point-to-point “interfaces” to maintain the scope definition

Where Hardware and Protocol Interfaces are Mismatched

- For some types of configurations one hardware interface may need to be represented by multiple protocol interfaces
  - Ethernet with VLANs
  - Frame Relay and ATM (i.e. relatively modern NBMA)
  - encapsulation-defined “interfaces” like PPPoE or tunnelling protocols
- For some interface arrangements several hardware interfaces may need to be treated as one protocol interface
  - Aggregated Ethernet
  - MLPPP
  - Bridging
- For some arrangements the relationship can be several to several
  - VLANs and PPPoE on aggregated Ethernet
- Making the relationship between hardware and protocol interfaces transparent is useful. While many VLAN “protocol” interfaces can share a single hardware ethernet there is still only one hardware down flag

(De)Multiplexing Protocol "Interfaces"

- The protocol interface of an incoming packet is recognized by a combination of the hardware interface it arrived on and some value in the packet's L2 header
  - For ethernet VLANs, the VLAN tag
  - For frame relay, the DLCI
  - For ATM, the VPI/VCI
  - For PPPoE, the ethertype, session ID and source MAC address
- To send a packet out a protocol interface requires encapsulating the packet with the appropriate L2 header and queuing it to the hardware interface output queue
- Each protocol interface sharing a single hardware interface is distinguished solely by an L2 header construction; there is still only a single output queue and input stream, that of the hardware interface
- While hardware interface configuration may be heavy weight, each protocol interface added to it adds only an additional L2 header variant. The marginal cost of adding a new protocol interface to existing hardware should be cheap

What BSD Provides to Deal With Interfaces

- An interface is represented by a `struct ifnet`. Hardware, protocol and pseudo interfaces all need to be represented by a `struct ifnet` since that is all there is.
  - A `struct ifnet` is quite large since it always includes the locking and queue structures, and procedure handles, required for a hardware interface
  - The fact that while there may be many protocol-level interfaces there is only one piece of hardware is opaque. There are many hardware down flags, many physical layer configuration variables and many output queues.
- For anything beyond the most basic configuration this is exceedingly messy.

What BSD Provides to Configure Protocols on Interfaces

- To enable a protocol on a `struct ifnet` you add an address for that protocol to its list of addresses. This has problems:
  - Some protocols an interface might be configured for, like IEEE 802 bridging, have no address to configure and need special casing.
  - IPv4 doesn't need interface addresses in some cases, like point-to-point links between routers, but there is no way to enable the protocol without them. It also wants an interface enabled for IPv4 without addresses to run a DHCP client to learn the addresses to configure (dhcpcd works around this by sending and receiving raw packets with bpf instead, a sure sign the protocol stack is missing something).
  - IPv6 has the opposite problem. It never needs to run without an interface address since it can (must?) make one up and add it to the interface automatically. Since adding an address is how you tell it you want a protocol enabled, adding one automatically leaves you no way to tell it when you don't want IPv6 enabled, adding another special case.
- This also ignores the fact that there may be other protocol-specific things (in and out firewall filters, a routing instance, a protocol MTU, procedure handles) to be configured when a protocol is enabled on an interface even if you don't have an address.

Hierarchical Interface Configuration

- A happier result can be obtained by splitting the `struct ifnet` into components that can be assembled into an interface configuration in a bigger variety of ways. The interface configuration components are:
  - The `ifdevice_t`, or `ifd`. Represents the configuration of a thing that has an output queue, i.e. the hardware view of an "interface".
  - The `iflogical_t`, or `ifl`. Represents the protocol view of an "interface", and the L2 values associated with that view.
  - The `iffamily_t`, or `iff`. Configures a particular protocol (IPv4, IPv6, bridging, …) onto an interface.
  - The `ifaddr_t`, or `ifa`. Configures addresses onto a protocol family.

The `ifdevice_t`

- The `ifd` is the structure autoconf will add when it finds appropriate hardware configured in a system.
- Holds configuration related to media selection, framing and clocking, i.e. the things needed by anything with a connector.
- Has a “down” flag representing the state of the hardware.
- Has an MRU, the maximum frame size that can be received and sent
- Has hardware-related procedure handles and an output queue
- Has an “encapsulation” selector (needed for HDLC interfaces) which defines what is acceptable to configure on top of this.
- Two types of structures may be configured as children of an `ifd`:
  - Zero or more `ifd`’s (this is probably no longer needed)
  - Zero or more `ifl`’s
- Naming is, e.g., wm%u, the same as a `struct ifnet` that might sit in the same place

The `iflogical_t`

- An `ifl` can be added as the child of an `ifd` or another `ifl`.
- Has an L2 identifier defining how packets for this ifl are distinguished, e.g. a VLAN tag for the child of an ethernet `ifd` (NULL is acceptable) or a VPI/VCI for the child of an ATM ifd
- Has a type. A PPP `ifl` might be the child of an HDLC interface or an ethernet `ifl` (for PPPoE).
  - The parent structure is the arbiter of good taste for the types and L2 identifiers a child may use.
- Has a "down" flag for use, e.g., to reflect the state of PPP LCP. Also collects the state of "down" flags in parent structures.
- Two types of structures may be added as children:
  - Zero or more `ifl`'s
  - Zero or more `iff`'s
- Naming is the name of its parent structure with a .%u appended, e.g. "wm0.2". If an `ifl` name omits the .%u suffix, .0 is assumed.
- Also has the `if_index` as an alternative name

The `iffamily_t`

- An `iff` can be added as the child of an `ifl`.
- A protocol family is configured “up” on an interface (to the extent possible without address configuration) when an `iff` for the family is added to the interface (but not before that).
- Has a “type” indicating the protocol family, e.g. IPv4, IPv6, bridging. Only one `iff` of a type can be configured on the same `ifl`, and not all `iff` types are compatible
- Has a routing instance identifier, i.e. selects a table to do route lookups for arriving packets (more later).
- Has input and output firewall filter specifications for forwarded and locally received packets
- Has the collected “down” flags of its parents
- Has an MTU for the protocol. This is configured (with a standard default) but needs to fit in the MRU of the parent `ifd` when the L2 header overheads are added in.
- Has procedure handles for, e.g. multicast operations
- Zero or more `ifa`’s can be added as children
- Is named by the name or `if_index` of its parent `ifl` plus the protocol type

The `ifaddr_t`

- An `ifa` is added to an `iff` to configure a protocol address for that family on the interface
- `ifa` configuration adds routes to the routing table as a side effect (or maybe as the main effect)
- There may need to be additional address preference configuration (if you have 2 addresses on the wire and send a multicast which is used? What about a unicast?)
- More or less the same as the current `struct ifaddr`

Use of This Configuration When Sending Packets

- The job of sending a packet is to find the L2 header to prepend to the packet and the output queue to add the now-encapsulated frame to
  - An L2 header associated with any bit of structure can be determined by asking the structure's parent for its L2 header and adding the structure's own contribution to it. The structure's children can recursively add their own contribution, allowing complex configurations to be built in simple increments
  - An `iff` knows its own protocol identifier and can get the remainder of the L2 header from its parent, maybe leaving only the L2 station address to be filled in from routing
  - An `iff` also knows where to queue the packet: to the queue associated with its parent `ifd`

Use of This Configuration When Receiving Packets

- The job of receiving a packet is to determine the (logical) interface the packet has arrived on and a function to call to process packets of that type.
  - The parts of the L2 header needed for this determination can be extracted from an incoming packet without reference to interface configuration; it's, e.g., the VLAN tag(s), if any, the ethertype and, if the ethertype is PPPoE, the session ID and source MAC address. This is a key for a search.
  - An iff knows the value of this key for its interface and protocol. When an iff is configured it can add it that to a search structure along with the `if_index` for its interface and a function that knows what to do with packets like that.
  - The driver looks up its key in the search structure. If it finds a match it knows the incoming interface and a function to call to deal with the packet. If it doesn't find a match the packet wasn't for us
- This procedure eliminates references to interface structures from the packet handling code, and deletes the big protocol case statement and knowledge of protocol types in general in favour of a generic key extraction and search
- Search structures can be modified without blocking lookups
- The elimination of protocol type knowledge from the driver makes support for L2 sockets which bind to L2 types possible, which enables support for a variety of protocols

Why This Is Good

- It provides a uniformity of configuration. A PPP `ifl` looks the same, and its child configuration is the same, whether it is configured on an HDLC interface's `ifd` or a VLANed ethernet's `ifl`. An ethernet `ifl` looks the same whether it is configured on an ethernet `ifd` or a PPP `ifl`
- The most common (by far) basic ifd→ifl→iff→ifa configuration can be easily made compatible with current procedures; just create the extra structures when an address is added
- When complexity is necessary, however, it provides a toolbox that can be used to build that complexity bit by bit to produce an attractive result
- Scalability is improved, not by the interface structure itself, but by using them in a way which keeps the packet handling path from having to touch them

The BSD Route Table

- There isn't just one BSD route table, there's a whole bunch
  - There's a table which is consulted to determine whether an incoming packet is addressed to a local unicast address
  - If not, there's another bit of code which is used to determine whether the packet is addressed to one of several broadcast addresses
  - If not, a "route cache" is consulted to see if it knows what to do with the packet
  - If not, the radix trie forwarding table is called to see if it knows what to do
  - Oh, and there's the multicast bag-on-the-side which looks up the same address in several different tables instead
  - Not to mention SO_DONTROUTE packets for which the interface address configuration can be used as yet another route table (an extremely slow lookup if you have a lot of interfaces)
- Each of these uses more-or-less the same data to key the search: an interface and the destination address, with the source address appended for the multicast lookup. Why do they all exist?

Unifying Route Lookups

- All of these lookups perform the same function. They take address and interface information and use that to determine what should be done with a packet that looks like that.
- What will be done with the packet falls into three general categories:
  - Add an L2 header to the packet and queue it to an interface (for multicast or bridging maybe more than one)
  - Receive the packet locally to see if any local applications are interested in it
  - Chuck the packet
- There is no reason to look at the destination address to see if it is multicast and, if so, do the lookup in a separate table
  - The route lookup looks at the same address and will return an appropriate result for multicast
- There is no good reason not to keep local and broadcast address routes in the same table as everything else for a single route lookup to find
  - Well, there may be structural reasons but it would be better to address these by fixing the structure rather than by breeding more route tables
- I don't believe there are inherent performance reasons to avoid full route lookups
  - If the full route lookup is too slow for you it is better to spend code improving its performance than it is to spend code inventing ways to avoid it
- It would be better if lookup structures were lockless and that's easier if there's only one

Route Tables Versus Forwarding Tables

- The issue with SO_DONTROUTE requires a bit of understanding of this
- There can be more than one way to reach the same destination, though different interfaces or via different neighbours.
- That is, there can be more than one route with the same address prefix, with the routes being distinguished by metadata. Routing protocols often learn several (or many) routes to the same place
- A route table (or RIB) holds all the routes you know, and provides lookups using both address information and metadata to find matches. Multiple routes with the same address prefix can coexist in a route table.
- For packets arriving from an interface you know the interface and the addresses in the packet, but you don't know any metadata (ignoring SSRR options) so there is nothing to distinguish between routes to the same destination
- A forwarding table (or FIB) hence holds only one route per address prefix since that is all packet forwarding can use

The SO_DONTROUTE Issue

- A SO_DONTROUTE lookup is a route lookup with metadata; it wants to find a particular kind of route directly to an attached neighbour, perhaps out a specific interface.
- Next hop resolution when installing routes requires an identical lookup, as does the SSRR option
- The unified solution to this is to always store interface routes in the route table and do a metadata-sensitive lookup to find the kind of route you want
- This requires storing more than one route per address prefix in some (fairly rare) cases:
  - More than one interface is configured on the same network
  - The user wants packets to the directly attached network routed indirectly instead
- Routes in the present BSD kernel are stored in forwarding tables; none can store more than one route to the same address
  - It hence can't assume all interface routes are present, even though they usually are
  - The local and broadcast address special case lookups have the same root cause
- While router forwarding paths can get by with a forwarding table, hosts and hosts-in-routers need a route table.

Performance

- The common argument for a forwarding path route cache is performance, that is most lookups can be satisfied in a simpler, smaller structure
- That this a net win relies on assumptions about traffic locality, and I don't know why those assumptions need to be true (though they often are for benchmarks…)
- The other assumption is that a full route lookup must be so expensive that a significant improvement in overall forwarding performance can be had by avoiding it, and this I dispute. We know more about longest match route lookups than when the BSD code was devised
- I actually have data for this (its not quite all talk). A more modern route lookup implementation may be available at http://www.netbsd.org/~dennis/rttree.tar.gz
- This implements a route table and can be modified with concurrent readers
- Scaling with table size may approach O(log(log(N)))
- Lookups were done to visit each route in a 533,000 route IPv4 table in two orders, a "bad cache" and a "good cache" order, with a 2.5 GHz Intel CPU. The results:
  - "bad cache" - 150 ns average per lookup
  - "good cache" - 30 ns average per lookup
- A forwarding path route cache would only improve on the latter number. On a machine where getting a packet from interface to interface in 1 us would be good this makes no difference

IP Multicast

- I not a fan of IP multicast. I like the service it offers but dislike what has to be done to provide it
- Never-the-less it is a standard part of the protocol suite, is in use and is certainly not different enough from the rest of IP to justify its bag-on-the-side
- Most of the “odd” things multicast forwarding requires have analogs in unicast or broadcast handling
  - Multicast forwarding routes are a prefix of the concatenated packet destination and source addresses. Just providing both addresses to the route lookup is probably as convenient as checking the address
  - Multicast next hops are lists of interfaces. Bridging next hops are too
  - Multicast needs an interface index in the route for an incoming interface check. Scoped unicast and broadcast addresses can use this too
  - Multicast forwarding routes are maintained as a cache (below 224/4), just like ND and ARP (below the subnet prefix). They can share a mechanism.
- I think the best reason to want it integrated in the route lookup is that it encourages one to think about “what would multicast do” when developing new functionality. I believe you get a better overall result this way no matter what you think about it

What's In A Route

- I hope I've persuaded you that wanting all address recognition done in a single table might not be mad. I can briefly describe what might be in a route
- Routes need to include the address information and prefix length, and the overhead for the structure they are stored in
- A route points at a next hop structure. All routes with the same next hop point at the same structure. This split is prompted by the fact that the number of next hops is O(#_of_neighbours) while the number of routes can be O(size_of_Internet). The latter is often much bigger than the former so it is better if routes are lean
- There needs to be a preference algorithm to choose the route to use for forwarding. This doesn't have to be fancy since multiple routes to a destination are rare and what you need to happen is usually either pretty clear or doesn't matter
- There is an interface index in the route. For routes which depend on interface address configuration it is that of the interface they depend on, otherwise it is passed by the user process and is used for an incoming interface check. There are flags to indicate when an incoming interface check should be done and what kind to do. The interface index is used as the secondary sort value by the preference algorithm.

Route Types

- The route type is the primary sort value for the preference algorithm. There are a small number of route types, described in preference order.
- User routes. These are (most of the) routes added from user space and are preferred over anything else. Only one user route with a particular destination prefix can be installed since there is no reason to do otherwise; since user routes always win a spot in the forwarding table it never needs to do a metadata search to find them.
- Permanent routes. These are routes which must be present in an otherwise empty table for standard address recognition to work properly, and exist at table creation. For IPv4 these are 0/0 (so a route lookup can always return something), 0.0.0.0/32, 224/4, 224.0.0.1/32, 255.255.255.255/32 and maybe 240/4.
- Interface routes. These are routes which are added either directly as a consequence of interface configuration, or subsequently by protocols like ARP or ND or by the user. When an `ifaddr_t` is added to a multiaccess interface 4 routes are immediately added, for a point-to-point interface 1 or 2. There are actually three separate types for interface routes to get the preferences to work right:
  - Interface local routes. Each of these is a local addresses from an `ifaddr_t`
  - Interface destination routes. Each of these is a destination prefix from an `ifaddr_t`
  - Interface routes. These are everything else
- Clone routes. These track the state of their forwarding parent in the table. Mostly used for multicast.
- There are also “ignore” versions of most route types to answer complaints about not being able to delete interface routes. These lower preference and make the routes unavailable for forwarding.

Next Hops

- The next hop is the useful result of the route lookup
- Next hops are reference counted, and are generally deleted when no references remain
- The next hop has a type which describes what it will do with the packet. It is used by user space to indicate the kind of next hop a route should have and the data that needs to be included in that specification
- The next hop has an output function. A packet is delivered by calling the output function with the next hop, the packet and an indication of whether the packet was received from an interface or originated by the local host
- The complexity is that the next hop output will need to do different things depending on whether the packet was received or originated
  - Received packets have not yet been forwarded (i.e. checked for options and had the TTL decremented, or dropped), while originated packets don't need to be
  - This is the structural change that enables route table unification
- In general, however, doing a full classification before the packet forwarding simplifies the work in each individual case even as it multiplies the places where this needs to be done. There's seldom a free lunch…

Routing Instances

- Now that we've packaged all routing information in just one spot it becomes convenient to have more than one route table
- Each routing instance is assigned an index. If you have only one table it is index 0, which is also the default when a table index is not specified, so all this is transparent when you only have one
- Each `iffamily_t` is configured with a routing instance. This is the table it adds its interface routes in, and is where the route lookup of packets arriving on the interface is begun
- When a packet is received by a local application the routing instance associated with the packet is the one that recognized the packet as being local. This may not be the same table that the lookup began in, but is the table where lookups for replies will begin
- Addresses now need to be qualified by routing instance, e.g. in socket bindings.
- Service sockets with only a port binding can receive packets from any instance; connections they accept will be bound to the instance of the incoming packets. Service sockets can also be bound to a particular instance (and will certainly be if the socket binding includes an address) to only receive packets from that instance
- A socket option allows an unbound socket to be associated with an instance before binding or connecting it, otherwise you get 0

Routing Instance Uses

- Routing instances are needed by routers to implement VRFs.
- Routing instances provide an interesing way to structure a NAT implementation, by separating “inside” and “outside” routes into separate tables. This allows you to have more than one “inside”, even if the addresses overlap
- For a host this can address an occasionally occurring problem: if you have 2 Internet providers and one host with an address from each provider, how can you route the return packets to the same provider’s router that the incoming packets arrived from?
  - Assigning the providers to different instances allows you to have two default routes (if the routers must be on the same interface, however, you could invent a new next hop type instead…)

Interface Dynamics and the Interface Index

- We've moved from having forwarding path interest in interface structures being directed at the relatively stable `struct ifnet`, which comes and goes only when hardware does, to the `iffamily_t` structure, which can be added and deleted at a whim
- This requires discipline concerning what is allowed to point directly at an interface structure (this discipline should be observed even now since hardware does come and go too). In particular the only structures allowed to point directly at an interface structure are those which will go away when the interface does
- Fortunately, there are two such structures in about the right spot. The search structure used to demultiplex to logical interface and protocol depends on the `iff` of the incoming interface while the next hop depends on the `iff` of the outgoing interface
- Every other reference to an interface is stored in the form of an `if_index`. This includes the most egregious offender, packets. The `if_index` may be dereferenced to find the interface but this operation is allowed to return NULL.

Centralized PCB Demultiplexing

- I am also a fan of centralizing PCB demultiplexing, that is of determining the protocol session and socket for a packet directly with a single lookup using the protocol 5-tuple (or 6-tuple counting an instance ID, or 7-tuple counting the incoming interface index), rather than demultiplexing to the protocol and having each do its own thing
- The reason is that it is useful for this lookup to be done in a lockless structure and this is easier if there is only one of them
- If you can get all the way from the interface to an individual protocol session without (signficant) locking then the locking you do have to do becomes very fine grained.
- In fact I actually prefer a different strategy: Rather than processing the packet all the way to the socket in this thread, instead queue the packet to a per-protocol-session queue and schedule the high layer protocol processing to be done by a thread running on the CPU of the likely receiver of the packet's data
  - This gives you parallelism in the network stack even when the network card only interrupts a single core, and allows the core being interrupted to spend most of its time doing what it must do, getting packets from the interface and figuring out who wants them
  - This is debatable, though, so I won't push it further

The PCB Lookup Key

- The PCB lookup is keyed by the following fields:
  - IP protocol
  - Local port (or maybe SPI for IPSEC)
  - Remote port
  - Routing instance index
  - Local address
  - Remote address
  - Incoming interface index
- Any field may be wild-carded. Nodes in the search structure have a mask indicating which fields must be matched
- There are fairly well defined best match semantics. In addition to entries for bound TCP sockets you can have an entry for the TCP protocol alone whose “protocol processing” consists of sending a port unreachable.
- The incoming interface index match allows one to bind a socket to receive only packets which arrive on a particular interface. This can be useful for routing protocols, or for DHCP
- When a socket is ready for use it inserts an appropriate entry to attract packets to itself
- The structure I have is a hybrid hash table/tree. It allows modifications to be with concurrent readers; some additions to the structure can be made without locking

What To Notice About This

- We mentioned a lookup at the input interface to determine the logical interface, a route lookup and a PCB lookup.
- The described forwarding path manages to get a packet from the input interface to an output interface or protocol session accessing only data in or directly pointed to by the results of lookups in those 3 data structures, all of which can be modified while being read
- You can delete interface configuration by removing the referencing structures from those tables and waiting until all operations that could be holding a late reference to them have finished. Once that's done you are free to delete the interface configuration.
- No packet processing needs to be blocked to do this. If you structure the packet processing path well you don't need locks
- The use of appropriately scalable data structures for searches accommodates large numbers of routes and a large interface configuration.

Conclusions

- We've had a lot of experience with the interface structures and have found that it can be "encouraged" to handle a big variety of ugly things, while being suitably unobtrusive when the configuration is simple
- Interfaces are software. You have have a lot of them
- The route table arrangement, with all address recognition being done in one spot, cleans up many warts, keeps all address-related operations efficient and avoids packet processing pawing through interface address structures.
- The separate next hop structure allows much invention.
- It is possible to have a very clean, scalable MP forwarding path if you maintain discipline over what is touched in that path.
- If you want more try http://www.mistimed.com/home/BSDNetworking.pdf